First Leaves for the Macintosh®

First Leaves for the Macintosh®

A Tutorial Introduction to Maple™

First Leaves for the Macintosh®

A Tutorial Introduction to Maple™

Bruce W. Char
University of Tennessee

Keith O. Geddes
Symbolic Computation Group
University of Waterloo

Gaston H. Gonnet
Symbolic Computation Group
University of Waterloo

Benton L. Leong
Symbolic Computation Group
University of Waterloo

Michael B. Monagan
Symbolic Computation Group
University of Waterloo

Stephen M. Watt
IBM Thomas J. Watson Research Center

Brooks/Cole Publishing Company
Pacific Grove, California

Brooks/Cole Publishing Company
A Division of Wadsworth, Inc.

COPYRIGHT
© 1989 by Wadsworth, Inc., Belmont, California 94002. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transcribed, in any form or by any means—electronic, mechanical, photocopying, recording, or otherwise—without the prior written permission of the publisher, Brooks/Cole Publishing Company, Pacific Grove, California 93950, a Division of Wadsworth, Inc.

MAPLE package design—© 1989 by Brooks/Cole Publishing Company

MAPLE: Symbolic Computation for the Macintosh—© 1988 by Symbolic Computation Group, University of Waterloo, Waterloo, Ontario.

TRADEMARKS
Maple is a trademark of the University of Waterloo. The Maple logo is a registered trademark of the Symbolic Computation Group, University of Waterloo.

Macintosh is the registered trademark of Apple Computer, Inc.

SOFTWARE LICENSE NOTICE
The software described in this manual is furnished under a license agreement (included in the package) and may be used or copied only in accordance with the terms of that agreement.

DISCLAIMER
The authors and Brooks/Cole Publishing Company (the publisher) make no representation or warranty with respect to the adequacy of this documentation or the programs it describes for any particular purpose or with respect to its adequacy to produce any particular result. In no event shall the authors or publisher of this documentation be liable for special, incidental, consequential, direct, indirect, or other similar damages arising from breach of contract, negligence or any other legal theory, even if the authors, publisher, or our agent has been advised of the possibility of such damages.

This product is neither endorsed nor coproduced by Apple Computer, Inc.

APPLE COMPUTER, INC. MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, REGARDING THE ENCLOSED SOFTWARE PACKAGE, ITS MERCHANTABILITY, OR ITS FITNESS FOR ANY PARTICULAR PURPOSE. THE EXCLUSION OF WARRANTIES IS NOT PERMITTED BY SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY PROVIDES YOU WITH SPECIFIC LEGAL RIGHTS. THERE MAY BE OTHER RIGHTS THAT YOU MAY HAVE WHICH VARY FROM STATE TO STATE.

Printed in the United States of America
10 9 8 7 6 5 4 3 2

ISBN 0–534–10222–0

Sponsoring Editor: Robert Evans, Jeremy Hayhurst
Marketing Representative: Peter Jackson
Editorial Assistant: Jennifer R. Greenwood, Virge Kelmser
Production Editor: Nancy L. Shammas
Manuscript Editor: John Bergez
Permissions Editor: Carline Haga
Interior and Package Design: Vernon T. Boes
Art Coordinator: Sue C. Howard
Compositor: Arthur Ogawa
Printing and Binding: Malloy Lithography, Inc., Ann Arbor, MI

SUPPORT
Customer support is available to registered users between 9 a.m. and 4 p.m. Pacific Time from Brooks/Cole Publishing Company. Call (408)373-0728 or AppleLink D2248 or Compuserve 70014,2461 for information on those services.

Preface

Maple™ is a system for symbolic mathematical computation. It is the ongoing product of a research project at the University of Waterloo and is currently used by thousands worldwide. It is designed for the computing learned in high school algebra and university mathematics—from fractions, factoring of polynomials, and exact solution of equations through calculus (limits, derivatives, exact solutions to indefinite and definite integrals and differential equations, and power series) to matrix algebra and beyond. Maple can also be used for numerical calculation, and for plotting functions. Scientists, mathematicians, engineers, and students all can benefit from the automated mathematics Maple makes available.

First Leaves for the Macintosh® is a tutorial introduction to Maple version 4.2 on the Macintosh. Chapter 1 explains how to install the Maple system on your Macintosh and how to use such features as file folders, windows, and the mouse while working with Maple. Chapters 2–5 present a tutorial that explains how to use Maple version 4.2 interactively as a calculator of formulas and numbers. For those who already have programming experience in another language, this manual also introduces how to use, write, and debug programs using Maple's programming language. Extensive examples are provided.

First Leaves is a complementary work to the definitive *Maple Reference Manual*, 5th edition, by Char, Geddes, Gonnet, Monagan and Watt (WATCOM Publications, 1988). As your needs and ambitions grow, the experience of working through the tutorial should make it easy for you to use the *Maple Reference Manual* (referred to in the text as *MRM*) and Maples's `help` command to learn how to use additional Maple features and library procedures that are not discussed in these pages.

Acknowledgments

We wish to express our appreciation to John Sellens, Ian Allen, and Ricardo Baeza-Yates for guidance in the ways of *troff*, *ms*, and LaTeX; to Mark Mutrie, Bev Marshman, Stan Devitt, and members of the Symbolic Computation Group of the University of Waterloo for their helpful comments on earlier drafts of this tutorial; to Bonnie Kent for editorial and text processing assistance; and to Arthur Ogawa. We also appreciate the many thoughtful suggestions submitted to us from the readers of the first version of *First Leaves*.

The Maple project has been supported, in part, by grants from the Academic Development Fund of the University of Waterloo, Digital Equipment Corporation, the Natural Sciences and Engineering Research Council of Canada, the Sloan Foundation, and the Information Technology Research Centre of Ontario.

Contents

Overview 1

CHAPTER ONE
Installing and Using Maple 5

1.1 Installing Maple on the Macintosh 5
1.2 Getting Started 8
1.2.1 Icons and files 8
1.2.2 Customizing your Maple program 9

CHAPTER TWO
Interactive Use of Maple 11

2.1 Starting a Maple session; how Maple behaves interactively 11
2.2 Simple arithmetic in Maple 11
2.3 Problems at the keyboard 13
2.4 Ending a Maple session; comments 14
2.5 Editing your Maple text 15
2.5.1 Reading text files 17
2.5.2 The Session window: Moving about and changing sizes 17
2.6 Evaluating expressions using enter 19
2.7 Computations 20
2.7.1 Progress cues 20
2.7.2 Stopping a computation 20

CHAPTER THREE
Computing with Maple 23

3.1 Maple variables 23
3.2 Printing values: `print` and `lprint` 25
3.3 Built-in commands for mathematical computation 26
3.4 Introducing Maple's mathematical commands 28
3.4.1 Differentiation 29

3.4.2 Indefinite and definite integration 30
3.4.3 Summation 31
3.4.4 Solution of algebraic equations 31
3.4.5 Other solvers: `dsolve`, etc. 36
3.4.6 Expression substitution with `subs` 37
3.4.7 Taylor and asymptotic series 39
3.4.8 Limits of real and complex functions 41
3.4.9 Numerical calculations to any desired precision 43
3.5 Other functions 45
3.6 Graphing and plotting functions on screen and on paper 46
3.7 Other formats for output: `fortran`, `latex`, and `eqn` 48
3.7.1 Fortran-style output 48
3.7.2 Output formatted for LaTeX and `eqn` 49
3.7.3 Maple session output to a file with `writeto` 50
3.8 What! You made a mistake? (Syntax errors) 50
3.9 You ask too much! (Run-time errors) 52
3.10 Saving the state of your Maple session 54
3.11 Recording results in files in human-readable format 56
3.12 Access to additional library procedures 56
3.13 `help` yourself to more of Maple 58

CHAPTER FOUR
Advanced Maple **63**

4.1 A few words to Fortran, Pascal, and BASIC programmers 63
4.2 Programming variables and mathematical symbols 64
4.3 Simplification of expressions 64
4.3.1 Automatic simplification 64
4.3.2 Simplifying expressions with `simplify` 65
4.4 Simplification with `expand`, `factor`, and `normal` 67
4.5 Full and delayed evaluation 70
4.6 Quotation and unevaluation 72
4.7 Using quoted variables as function arguments 74
4.8 Concatenation: Forming new names from old 75
4.9 Looking at parts of expressions: `op`, `nops`, `coeff` 77
4.10 Expression sequences, sets, and lists 82
4.10.1 Expression sequences 82
4.10.2 Sets and lists 84
4.11 Tables and arrays: Indexed collections of data 86
4.11.1 Tables 86
4.11.2 Arrays 90
4.12 Converting from one structure to another 91
4.13 The `map` function: Performing the same operation to all elements of a data structure 93
4.14 Linear algebra in Maple; the `with` function; packages 95

CHAPTER FIVE
The Maple Programming Language 99

5.1 Repetition `while` you wait 99
5.2 Repetition `for` each one 101
5.3 Conditional execution with `if-then-else-fi` 103
5.4 `break` and `next`: Control within `for-while` loops 105
5.5 Simple Maple procedures 106
5.6 More Maple procedures: Multiple statements, local variables, `RETURN` 108
5.7 `Option remember`: Remembering previously computed function values 113
5.8 Using `ERROR`: Exiting several procedures at once 114
5.9 Packages in Maple 117
5.10 Your Maple initialization file 118
5.11 File Names 119
5.11.1 Folder name 119
5.11.2 Name translation 120
5.12 Creating your own library 120
5.13 Monitoring time and space consumed during a computation 123
5.14 Garbage collection and `gc` 125
5.15 Debugging Maple programs: `printlevel` and `trace` 126
5.16 Conclusion 130

APPENDIX
Symbols with Predefined Meanings in Maple 131

Index 135

Examples

1 Potpourri of algebraic calculations in Maple 1
2 Fibonacci procedure 3
3 What you see after you type `1 + 1;` and hit the enter key 11
4 Simple numerical calculations 12
5 Problems that occur when you don't use parentheses liberally 14
6 Assignment of values (labeling) 24
7 `print` and `lprint` 25
8 Assortment of automatic mathematical simplifications 26
9 Maple calculations with mathematical unknowns 28
10 First derivative with respect to x via `diff` 29
11 Partial derivative with respect to s and t 29
12 Symbolic indefinite and definite integration with `int` 30
13 Summation with `sum` 31
14 Solution of one equation and one unknown 32
15 Solution of system of equations 33
16 What happens when no solutions, or only some solutions, are found 34
17 Solving differential equations with `dsolve` 36
18 Substitution of an expression for another via `subs` 38
19 Simultaneous substitution of a set of expressions 39
20 Taylor and asymptotic series calculations 40
21 Real bidirectional limits 41
22 Other directional limits 43
23 Floating-point numbers in Maple 43
24 Approximating exact results with `evalf` 45
25 Plot of a single function 46
26 Fortran output displayed on the screen 49
27 Generating a LaTeX version of a matrix 49
28 A collection of syntax errors 50
29 What happens when your backquotes don't come in matching pairs 51
30 A collection of run-time errors 52
31 Retaining Maple results between sessions 55
32 Reading in a library procedure with `readlib` 57
33 Overview of on-line help via `help()` 58

34 help for library functions 59
35 help index of general topics; help on Maple statements 59
36 Getting help for the reserved word done 60
37 Examples of the simplify command 66
38 Expansion of multiplication over addition, and of cos, exp 68
39 Using normal to simplify rational expressions 69
40 Factorization of polynomials over the integers via factor 70
41 "Full evaluation" follows the chain of assignments 70
42 A problem due to full evaluation 71
43 The effect of evaluation, levels of quotation 72
44 Assigning and unassigning variables 73
45 Resolving the problem with i and sum 73
46 Use of quoted variables as parameters to library procedures 74
47 Concatenation of symbols into a name 75
48 Types and parts of expressions via whattype and op 79
49 Finding coefficients of variables with coeff, lcoeff, and tcoeff 81
50 Sequences and the $ operator 83
51 Set operations 84
52 Merging lists, reversing a list 85
53 Creating and modifying tables 86
54 Another example of a Maple table 87
55 Tables as assigned values, last-name evaluation 88
56 Copies of tables 89
57 Maple arrays 90
58 Converting between data structures 92
59 map applies a function to each component of a structure 93
60 map with more than two arguments 95
61 The linear algebra package 95
62 More linear algebra, without with 98
63 Use of while to compute integer gcd's: printlevel 99
64 for statement 101
65 for-in statement 102
66 Using if to print out results of primality testing 103
67 if ... then ... elif ... fi 104
68 Leaving a for repetition early with break 105
69 A simple Maple function defined and used 107
70 A simple multistatement procedure defined and used 108
71 Evaluation of local variables 109
72 Procedure using local variables 110
73 A procedure to compute Chebyshev polynomials using RETURN 111
74 The meaning of names when one procedure is defined within another 112
75 A function remembering a special value 114
76 Using the ERROR statement to signal a computational exception 114
77 lasterror and traperror 115

78 Trapping an expected error 115
79 Nonlocal returns using ERROR/traperror 116
80 A simple package 117
81 Using your own package 118
82 With a package from your library 121
83 Setting up a readlib-ed package 122
84 A member of a readlib-ed package 122
85 Computing elapsed time with the time function 124
86 Programming aids: printlevel 127
87 Programming aids: trace and untrace 128

Figures

1 Example of a plot produced with Maple 2
2 Maple folder after installation 7
3 Maple icons 9
4 Customization dialog box 9
5 Display of 100! 16
6 The **Edit** menu 16
7 Wide expression fitting across a window 18
8 Wide expression now partially out of view 18
9 The **Font** menu 19
10 The **Halt** menu 21
11 "Out of stack space" dialog box 53
12 The **Status** window 123

Tables

1 Arithmetic operations in Maple 12
2 Mathematical constants known to Maple 26
3 Mathematical functions known to Maple 27
4 Options used with `dsolve` 36
5 Taking directional limits 42
6 `plotdevice` values 47
7 Varieties of transformations available with `simplify` 66
8 Types of mathematical objects in Maple 78
9 General-purpose data structures in Maple 82
10 Set operations in Maple 84
11 Wavelength table 86
12 Matrix/vector operations 96
13 Symbols with predefined meanings in Maple 131

Overview

First Leaves for the Macintosh tells you how to use Maple as your "algebraic calculator," applying the speed and thoroughness of the computer to the algebra and numerical calculation you might have done by hand. Maple is useful for calculations involving formulas, algebra and calculus, matrices, the exact solution of equations, and more. Through Maple, you can draw easily and automatically upon the mathematical expertise and advances in computer science built into Maple during the past decade by its authors at the University of Waterloo and their colleagues.

Before we start our discussion of how to use Maple on your Macintosh, let's take a quick look at a few of the things you can use Maple to compute. The simplest way to use Maple after launching it, is to type an expression, follow it by a semicolon, and press enter. Example 1 shows how Maple performs the following algebraic calculations:

(1) Compute $\sum_{i=1}^{n} i^3$ and simplify the result.
(2) Compute $\int a(x-1)\cos a(x-1)dx$.
(3) See whether one polynomial divides another. Print the quotient.

Example 1
Potpourri of algebraic calculations in Maple

```
#Programmer's comments begin with a "#" and continue to the end of the line.
#They are not regarded by Maple as instructions to compute anything.
#
# Calculation 1
#
>
  sum(i^3, i=1..n);
                                  4             3             2
                1/4 (n + 1)^4 - 1/2 (n + 1)^3 + 1/4 (n + 1)^2

>
  simplify(");
                                     4       3       2
                              1/4 n^4 + 1/2 n^3 + 1/4 n^2

>
  expr1 := a*(x-1);
                                expr1 := a (x - 1)
```

```
# Calculation 2
#
>
  int(expr1*cos(expr1),x);
                  cos(a (x - 1)) + a (x - 1) sin(a (x - 1))
                  -----------------------------------------
                                      a

# Calculation 3
#
>
  divide(x^11-x^10+x-1, expr1/a, 'quotient');
                                      true

>
  print(quotient);
                                     10
                                    x   + 1
```

Glancing at the results of Example 1, we see that Maple can compute results that are *formulas*. It can also handle conventional numerical computation, even if the exact results are hundreds or thousands of digits long.

First Leaves for the Macintosh also shows how to use Maple for drawing graphs of functions, as in Figure 1.

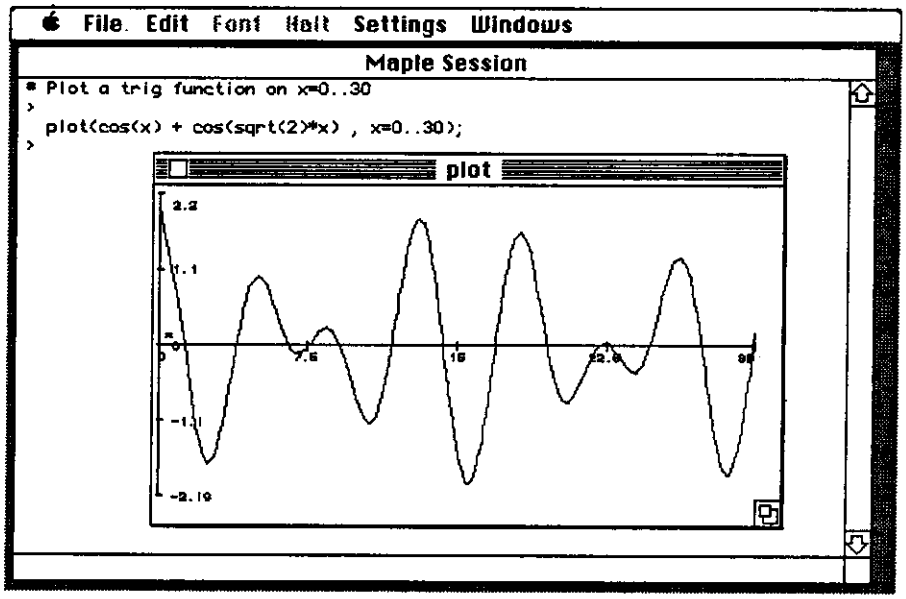


FIGURE 1. Example of a plot produced with Maple

As you progress through this tutorial, more involved computations using the Maple programming language are demonstrated, such as the computation in Example 2.

Example 2
Fibonacci procedure

```
#The following defines a short Maple program to compute the N-th Fibonacci
#number, where the N-th Fibonacci number is defined to be the sum of the
#previous two Fibonacci numbers.  Details of Maple's programming
#language, such as how to write procedures like this one, are discussed
#in Chapter 5.
#
>
   fibonacci := proc(N)
>
       option remember;
>
       if nargs<>1 or not type(N,integer) or N<0 then
>
           ERROR(`wrong number or type of parameters in fibonacci`)
>
       else
>
           if N<2 then N else fibonacci(N-1)+fibonacci(N-2)  fi
>
       fi
>
   end:
#Now compute the 10th fibonacci number by executing the program with N=10.
#
>
   fibonacci(10);
                                      55
#Compute a few values in the sequence.
#
>
   for i from 0 to 2 do
>
       lprint(`fibonacci(`, 10^i, `) is `, fibonacci(10^i))
>
   od;
fibonacci(   1   ) is    1
fibonacci(   10   ) is    55
fibonacci(   100   ) is    354224848179261915075
```

The basic Maple program, referred to as the *kernel*, can do many simple mathematical calculations involving numbers and formulas. When more complicated calculations are called for, it can draw on a library of mathematical procedures. The Maple library contains directions, specified in Maple's programming language, for doing more complicated things such as factoring polynomials, performing integration, or solving systems of equations. These library procedures are ordinarily kept in files on the disk storage of your computer, but most are automatically loaded into Maple when needed.

Together, the kernel and library contain Maple's built-in knowledge of mathematical computations. The commonly encountered Maple "built-in commands" are discussed in Chapters 2 and 3.

There are other mathematical operations for which no "prepackaged" programming exists. In such cases, step-by-step directions must be given to the computer to get the desired result. For example, there is no built-in command to rearrange an equation so that the occurrences of the symbol x and those of y are on different sides of the equal sign. Chapter 4 discusses the details of directing Maple step-by-step through an interactive manipulation. Chapter 5 describes the details of the Maple programming language. Even though many useful calculations can be performed without knowledge of programming, you may wish to program Maple so that it knows about mathematical functions not supplied by the library or to automate a laborious sequence of calculations.

This tutorial supplements the *Maple Reference Manual*, 5th edition, by Char, Geddes, Gonnet, Monagan, and Watt (WATCOM Publications, 1988). Throughout the tutorial, references to the *Manual* are abbreviated *MRM*.

CHAPTER 1

Installing and Using Maple

● **ATTENTION**

Before you attempt to install or use Maple on your Macintosh, it's important that you understand some of the common terminology associated with the Macintosh system. If you are a brand-new user of the Macintosh, you should complete the Macintosh tour that comes with your system. We assume you are familiar with common Macintosh terminology for the remainder of this tutorial.

1.1 Installing Maple on the Macintosh

Because the Maple system contains an extensive library of mathematical routines, you will need to install the software from the original floppy disks onto a hard disk. You will need about seven megabytes of free space on your hard disk in order to install Maple.[1]

To run Maple, you will need to have the Maple system installed on a Macintosh with one megabyte of RAM. The following installation procedure takes about an hour.

1. Insert the disk labeled `Maple Program Disk` into a floppy-disk drive. Drag the Maple disk icon onto your hard disk. You will see a warning about copying from one disk to a different type of disk. Since that is exactly what you want to do, click on the OK button. This step creates a folder labeled `Maple` on your hard disk.

[1] The Maple library contains Maple object modules that are loaded as necessary during a Maple computation. The library also contains the source files for each Maple object module. These source files are not necessary for the use of Maple, and you can delete them from your disk either during or after the installation process to conserve space on your disk. Deleting the source files will free one to two megabytes of disk space. Any folder with the name `src` within the Maple library folder or subfolders of the library folder contains Maple source code and can be deleted. You may choose to leave the source files installed on the disk, however, because they are informative and can show you the procedures that Maple uses to solve problems. They can also serve as programming models for advanced users who wish to write their own Maple routines.

2. Now eject the Maple program disk by dragging the program disk icon into the garbage can icon on your desktop. You will know that the disk has been placed in the garbage can, and not just near the garbage can, when the garbage can becomes highlighted as you drag the program disk icon over it.

3. Open your new Maple folder by double-clicking on the folder. There may be a file in the folder named `Read Me`. This file, if it's there, contains notes about your Maple system that have been added since this tutorial was printed. You can use the Teach Text program, which is distributed by Apple with its system software, to read this file, or you can use most other word processing programs to read it. *Do this now,* because the additional documentation may contain important changes regarding the rest of these instructions.

4. Inside the Maple folder, you should see two programs shown with the maple-leaf program icon. Maple Plus is the version that you should use for the Macintosh Plus and Macintosh SE. Maple II is the version to use for the Macintosh II and Macintosh IIx. Although Maple Plus can be used on all the models of the Macintosh mentioned, the Maple II version takes advantage of the more powerful processors of the Macintosh II and Macintosh IIx and is therefore more efficient. Maple II should never be used on the Macintosh Plus or the Macintosh SE.

Rename the appropriate program for your machine as `Maple` and remove the other program by dragging it into the garbage can. To rename the program, simply click once on its icon. This should highlight the icon on the screen. Now type the new name, `Maple`, and then press the return key.

5. Using the **New Folder** menu item from the **File** menu, create a new folder within the Maple folder. Name the new folder `Lib`. This folder will contain the Maple mathematical library routines.

6. Insert the first Maple library disk, labeled `Maple Lib 1`, into a floppy drive. Open the library disk by double-clicking on the disk icon. Slide the `Maple` folder window and the `Maple Lib 1` window around on the screen so that the `Lib` folder inside the `Maple` folder is not covered by another window. Activate the `Maple Lib 1` window by clicking anywhere within that window. Now use the **Select All** menu item from the **Edit** menu to select all items in the `Maple Lib 1` window. All items inside that window should now be highlighted.

7. Drag any one of the highlighted items into the `Lib` folder in the other window, and you should see the outlines of many document icons following your cursor into the `Lib` folder. A dialog box will display the progress of the disk copy operations as the Maple library files are copied in the `Lib` folder on your hard disk. When this operation has been completed, eject the `Maple Lib 1` disk by dragging it into the garbage can.

8. Repeat steps 6 and 7 for the library disks `Maple Lib 2` through `Maple Lib 7`, but not for `Maple Lib 8`.

9. The contents of `Maple Lib 8` are Maple on-line help files. These files are to be placed inside the `Help` folder, which is inside the Maple `Lib` folder. Open the Maple `Lib` folder by double-clicking on it. The library folder contains hundreds of files, so opening it will take longer than opening most other Macintosh folders. Since the folder does contain so many files, when it is opened, you may have difficulty locating the `Help` folder. If you use the **View** menu to view the folder **by Name**, you'll see

everything listed alphabetically and can easily scroll down the window to the `Help` folder.

10. Insert the `Maple Lib 8` disk into your disk drive and open that disk. By sliding the open windows on your screen, rearrange them so that the `Help` folder from the Maple `Lib` folder is visible. Activate the window for `Maple Lib 8` by clicking in that window and then select everything from that window by using **Select All** from the **File** menu. Now drag all the documents from the `Maple Lib 8` disk into the `Help` folder.

11. This completes the installation of Maple onto your hard disk. Eject the `Maple Lib 8` disk. Close the `Help` folder and then the Maple `Lib` folder. Your `Maple` folder should now look like the one shown in Figure 2.

To find the Maple internal-format files, which might be loaded for some computations, Maple needs to know where the Maple library is located on your disk. Maple uses the variable `libname` to point to the location of this library. As with any other Maple variable, you can change the value for `libname` as many times as you'd like during a Maple session. However, if the value is set incorrectly, then you'll be told that Maple can't find the library routines when it needs to load new ones. So once `libname` is initialized correctly, you should leave it alone. In the original copy of Maple you installed, the initial value assigned to `libname` is `:Lib`,

If it were necessary to place the Maple program in a different folder than the one containing the Maple library, then you would have to change the initial setting used for `libname`. Do this by using the **Initial library name...** item in the **Settings** menu. When you select this item, you see the current initial library name setting that

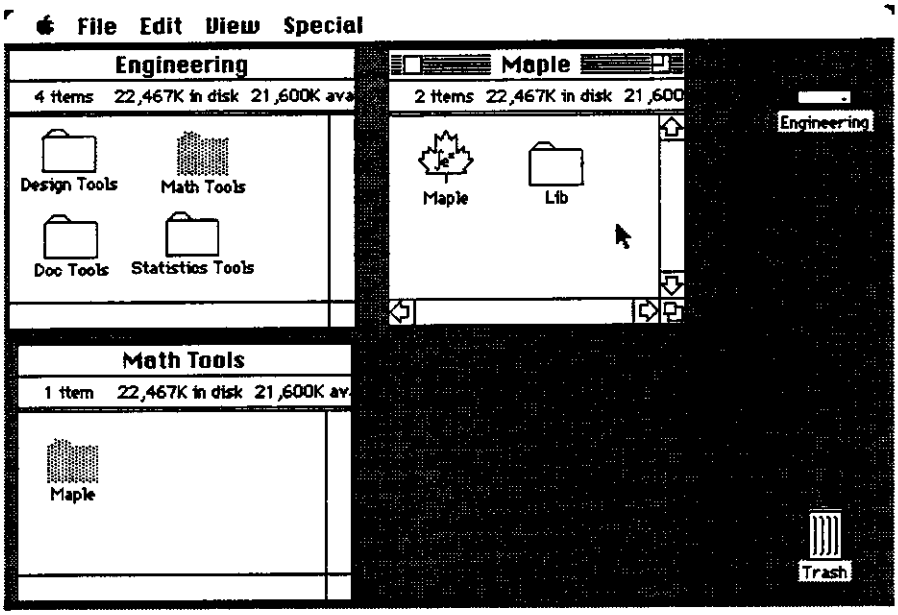


FIGURE 2. Maple folder after installation

you can edit so it is correct for your configuration. To easily verify that you have set the initial library name correctly, launch Maple and then type the `help();` command. Because this command needs to access the library to both load the help routine and also to read an on-line help page stored in the library, you'll find out immediately if the value you typed for the library name setting is correct.

● **ATTENTION**

The typing of volume names, folder names, and program names in a file pathname is often confusing, especially when there may be blanks in the names. We recommend that you leave both the Maple program and the Maple library in the same folder and use the default setting of `:Lib` for `libname`. For more information about specifying a pathname to your library, see Section 5.11.

1.2 Getting Started

1.2.1 Icons and files

You can start or launch Maple by double-clicking the Maple program icon itself or any of the documents created by Maple. These documents include text files, plot files, and binary (or object) files. You can recognize a Maple document type by its icon (see Figure 3).

A Maple *text file* (indicated by the maple-leaf icon with broken lines) can contain commands that you have typed during a Maple session, commands and comments that you have created with any other program on the Macintosh that creates a `TEXT` file, or a `TEXT` file imported from another computer. (Text files created by Maple as output can also be inserted into other programs, including word processors.)

A *plot file* (indicated by the maple-leaf icon with a graph) contains Macintosh `PICT` commands that draw a graph. These plot files can be created in one Maple session and then printed or saved for use in another session.

A Maple *internal-format file* (indicated by the maple-leaf icon with `10101`), also known as a *library object file*, contains commands that have already been read by Maple and that are stored in a way that is efficient for Maple to use. These files are created by Maple and should be read only by Maple. They are sometimes referred to as `.m` (pronounced "dot m") files, since Maple stores these files on many computer systems with file names that end with `.m`. The use of Maple internal format files is discussed in Section 3.10.

Now that we know that we can launch Maple on the Mac by double-clicking any Maple document or by double-clicking the Maple program icon itself, let's follow the simplest task of starting a Maple session by double-clicking the program icon.

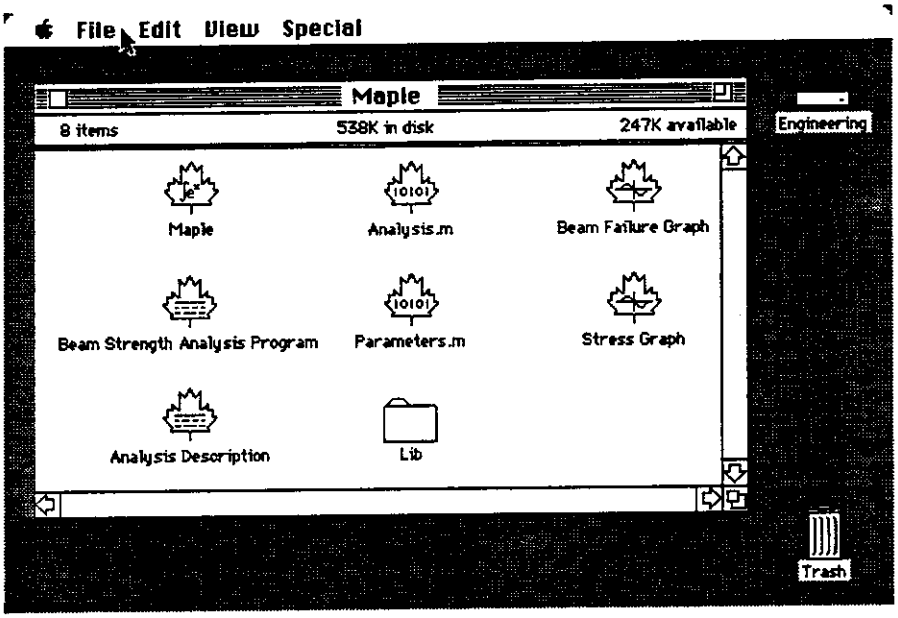


FIGURE 3. Maple icons

1.2.2 Customizing your Maple program

If this is the first time that you've used your Maple program, it will present the display shown in Figure 4, asking you to customize your version of the program.

Type your name and the name of your organization in the two boxes provided. This will be the only time that you will be asked to do so. After you have customized your copy of Maple, it will display a startup screen showing the information that you

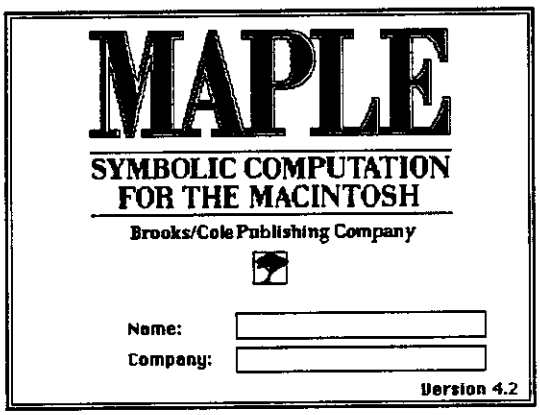


FIGURE 4. Customization dialog box

have provided and the version number for your copy of Maple. The startup screen will disappear by itself after a short while, but eager users will want to press any key or the mouse button to move right along into their Maple session.

Chapter 2 explains what to do once Maple has proceeded beyond the customization dialog box.

CHAPTER 2

Interactive Use of Maple

2.1 Starting a Maple session; how Maple behaves interactively

Chapter 1 describes how to start a Maple session. When you are successful in starting a session, the startup screen is displayed briefly and then the **Maple Session** window appears. The **Status** window behind the **Maple Session** window displays information about the amount of memory and time used by Maple and is explained in more detail in Section 5.13.

Maple is now ready for you to enter an expression or a command. Maple commands are simply expressions consisting of a call for a predefined Maple function. When the prompt (>) is displayed in the session window you normally type an expression on the next line. When expressions are entered, the cursor on the screen turns into a watch while Maple performs the calculation or evaluation. If several Maple statements are evaluated during the course of the calculation, the watch hand spins. When the calculation is completed, the cursor reverts to an I-beam and the results of the calculation are displayed in the session window. The prompt also appears once again. As you work with Maple, you will see what you type and the numbers and formulas that Maple generates in response to your commands.

2.2 Simple arithmetic in Maple

To add, subtract, multiply, or divide integers, fractions, polynomials, or rational functions, type in the expression you wish computed. Typically, when working interactively with Maple, you end such an expression with a semicolon followed by [enter]. On some Macintosh keyboards, this key is somewhere below the large [return] key; on other keyboards, it is on the extreme right side of the keyboard.

Example 3
What you see after you type `1 + 1;` and hit the [enter] key

```
> 1 + 1;
                              2
```

● ATTENTION

Don't forget the semicolon! This grammatical requirement of Maple is forgotten by many users doing their first Maple calculation. You can't progress to the second calculation without the semicolon ending the first.

Maple's symbols for arithmetic are shown in Table 1.

TABLE 1
Arithmetic operations in Maple

Operation	Symbol	Example
addition	+	2 + 2
subtraction	-	10 - x
multiplication	*	3 * y * z
division	/	x / 2
exponentiation	^ or **	x ^ 2 x ** 2
absolute value	abs(...)	abs(-4)
factorial	!	10!

After you enter an expression, Maple calculates the result—for example, figuring out that $2 - 3$ is -1. This calculation process is referred to as *evaluating the expression.* The result of the evaluation is displayed in two-dimensional mathematical notation (see examples that follow), or "prettyprinting." When directing Maple through a sequence of commands, you can refer to the previously computed result by means of the double-quote symbol ("), as in Example 1, where `simplify(")` is the command to apply certain built-in "simplifying" transformations to the previously computed result.[1] If you don't wish to display a result, you can end things with a colon (:) instead of a semicolon. Here are some simple arithmetic expressions computed in a Maple session.

Example 4
Simple numerical calculations

```
> 2 + abs(-2);
                              4
```

[1]The symbol "" (two double-quotes) stands for the next-to-last result, and the result before that is referred to by """.

```
> (4 + (" * 6))/(999999-32516);
                                    28
                                 ------
                                 967483

> 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10 * 11 * 12 * 13 * 14 * 15 * 16
>   * 17 * 18:
> " - 18!;
                                    0
```

2.3 Problems at the keyboard

It's not uncommon to make an occasional mistake in typing. If you notice a mistake in an expression before you have entered it, you can use the [delete] key to backspace to the mistake and then retype whatever is necessary. However, a simpler and often faster way to make corrections is to position the cursor next to the mistake and then click the mouse button. The blinking insertion bar will move to the cursor position. Use the delete key to remove characters; type extra characters to insert them. Then press [enter].

If you have already pressed [enter] by the time you notice your mistake, Maple may or may not evaluate your expression. If it happens to be a grammatically correct expression, in spite of the mistake, Maple will evaluate the erroneous expression. If the expression is not grammatically acceptable, Maple will display this message

```
Syntax error before:
```

and the portion of the expression following the point where Maple noticed the error. After entering an erroneous expression, you can move the insertion bar to the error, correct it, and then press the [enter] key to reenter it.

Notice that you don't move the insertion bar to the end of the line or expression after correcting a mistake and before reentering the expression. The reason is that when you press [enter] the entire line on which the insertion bar lies is read by Maple for evaluation. If you should ever type an expression on the same line as a prompt and then press [enter], the > prompt will be read as part of the expression. This will certainly produce a syntax error message. A common mistake is the use of the [return] key instead of the [enter] key. If you press [return] instead of [enter], it's not a terrible mistake. Just move the insertion bar up one line to the expression and then press [enter]. You can also use the [↑] key to move the insertion bar up a line before entering.

The reason you don't use the [return] key to enter expressions will become clear when we present Maple as a text editor in Section 2.5.

● ATTENTION

Parentheses should be used to be clear about the order of arithmetic—for example, `(5 + 4)*6` versus `5 + (4 * 6)`. The expression `-1^(1/2)` means `-(1^(1/2))` or -1, so if you wish to refer to the square root of -1, use parentheses: `(-1)^(1/2)`. The precision of meaning conveyed with parentheses usually makes up for the extra typing.

Users who omit "unnecessary" parentheses often encounter problems with the operation of negation and with Maple's grammatical requirement that $a^{(b^c)}$ and $(a^b)^c$ be written as `a ^ ( b ^ c )` and `(a ^ b) ^ c`, respectively. `a ^ b ^ c` is not a grammatically correct phrase in the Maple language. If you wish to write Maple expressions in the fewest number of parentheses, consult the *Maple Reference Manual* (MRM) or **`help(operators,precedence);`** for detailed descriptions of the precedence of operations.

Example 5
Problems that occur when you don't use parentheses liberally

1. Negation has a lower precedence than exponentiation:

```
> -5^2;
                                  -25
```

2. You must parenthesize negative terms according to the grammatical rules of Maple:

```
> -5^-2;
syntax error before: -2;
> -5*-2;
syntax error before: -2;
```

3. Negation has a higher precedence than multiplication, but lower than exponentiation.

```
> -5*(-2);
                                   10

> -5^(-2);
                                 -1/25
```

4. You must parenthesize towers of exponents

```
> 2^3^4;
syntax error before: ^4;
> 2^(3^4);
                  2417851639229258349412352
```

2.4 Ending a Maple session; comments

The command **`quit`**, followed by a press of the [enter] key, ends a Maple session. **`done`** or **`stop`** also will work.

Maple treats any line or portion of a line that begins with a # as a *comment*. The only thing that Maple does with a comment is to display it back:

```
>
  #Here is a comment        User types in a comment line
#Here is a comment          Computer types out comment line again.
>
```

In examples in this tutorial, you will often see descriptive comments interspersed with Maple commands. Maple comments are also useful to programming practitioners who read or write Maple programs, as is further discussed in Chapter 5.

● ATTENTION

In the examples in this tutorial, we have suppressed the comment line repetitions to make the examples more compact.

2.5 Editing your Maple text

You can use Maple to create any text window, or text file, you like. You can use it as a simple text editor without ever having anything evaluated by Maple—provided you don't use the enter key to enter a computation. To create a document containing many lines of text, use return. In Maple, return is not used to cause the program to evaluate an expression. It is used, however, in the same way that the return key is used when you enter text with many text editors.

To illustrate the simple editing of an expression, try a simple computation. Type `10!;` and press enter. Maple will compute 10 factorial by actually multiplying 10 by 9 by 8, and so on. The computation was fast, but then for such a small factorial we expect it to be fast. Let's try something a bit more challenging: 100!. We can create this expression by simply editing the previous one. Place the blinking insertion bar between the 1 and the 0 in the `10!` that you typed earlier by using the mouse to position the I-beam cursor at that point and pressing the mouse button. This allows you to edit the expression by inserting characters at this point. Insert an extra zero to change the expression to `100!`. Now simply press enter and 100! will be computed and computed very quickly (see Figure 5).

So far we have illustrated some very rudimentary editing that can be done with Maple. More sophisticated editing can be done through the use of the operations provided in the **Edit** menu (see Figure 6). By clicking at the beginning of an expression, dragging the cursor to the right until you reach the end of the expression, and then releasing the mouse button, you will have selected the entire expression, which should be highlighted on the screen. You can delete this expression by using the **Clear** command from the **Edit** menu, or you can place a **Copy** of it in the Clipboard without deleting the original occurrence. (**Cut** text is placed in the Clipboard as well.) Any text placed in the Clipboard can be inserted anywhere in the **Maple Session** window.

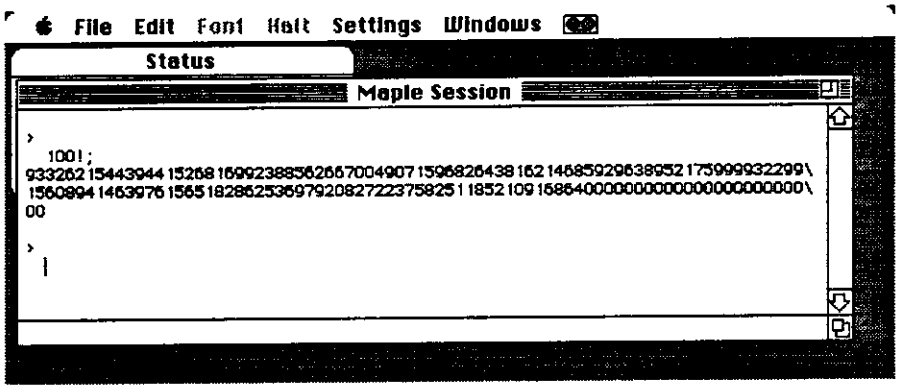


FIGURE 5. Display of 100!

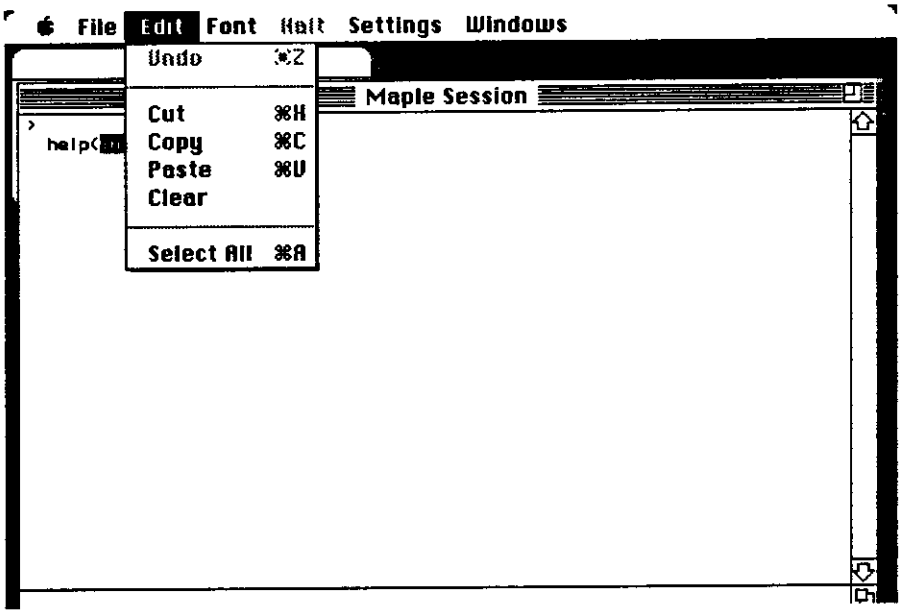


FIGURE 6. The **Edit** menu

Later, as you create Maple procedures, you will appreciate the ability to use Maple as a text editor to create a procedure, review it, and make changes to it before you finally enter its definition. For any but the simplest of procedures, the use of the return key to lay the procedure out on several lines is crucial.

2.5.1 Reading text files

The results of computations from previous Maple sessions or a file of text created by another program can be read into Maple in various ways. The simplest is for you to choose the **Open** item from the **File** menu and then select the text file to be opened. This will create a new text window in your Maple session containing the contents of the file. *None of the text in that file is assumed to be Maple expressions or commands, and nothing is automatically entered from that window to Maple for evaluation.* This gives you the flexibility of reading a documentation file into a window, such as a file containing your notes on a problem being solved along with the results that have been calculated using Maple or a log of a previous session.

You can select any text from the text window by normal text selection techniques using the mouse, or you can place the edit insertion bar onto any line in that window. Pressing the [enter] key will then enter the selection or line for evaluation, with the results of the evaluation appearing in the **Maple Session** window. You can also use the standard text editing techniques of cutting and pasting to move text back and forth between various text windows, including the **Maple Session** window.

2.5.2 The Session window: Moving about and changing sizes

As you enter more and more expressions for Maple to evaluate, the **Maple Session** window will be filled, and previous expressions will scroll up out of view. To review the history of your session or to bring back previous expressions for reuse, simply use the vertical scroll bar on the right to scroll previous expressions back into view.

If you want to make the window smaller—for instance, when there are other windows on the screen that you would like to see at the same time—drag the resize icon in the bottom right corner of the **Maple Session** window until you have adjusted the window to whatever size you would like.

Expressions displayed as the result of a computation by Maple will be formatted for the current window width (see Figure 7). Reducing the width of the window may force parts of previous expressions that were originally displayed in a wider window out of view (Figure 8). You can bring the expressions back into view by reenlarging the width of the window.

If Maple needs to break up a large name or a large number because of the width of the window, it will place a continuation character (\) at the end of the line to indicate that a name or number is being continued.

Another way to make the window effectively smaller or larger with respect to the display of expressions is to change the size of the font used in the window. Using the **Font** menu (Figure 9), you can select one of several sizes for the text in a text window. If the font size used as the default is too small for you to read comfortably, choose a larger font size. If you would like to see more information displayed at one time, choose a smaller font size.

The font sizes shown in the menu in larger outline characters are the natural font sizes contained on your system; these are the font sizes that have the best appearance. The other sizes are magnifications or reductions of these natural sizes. As you change

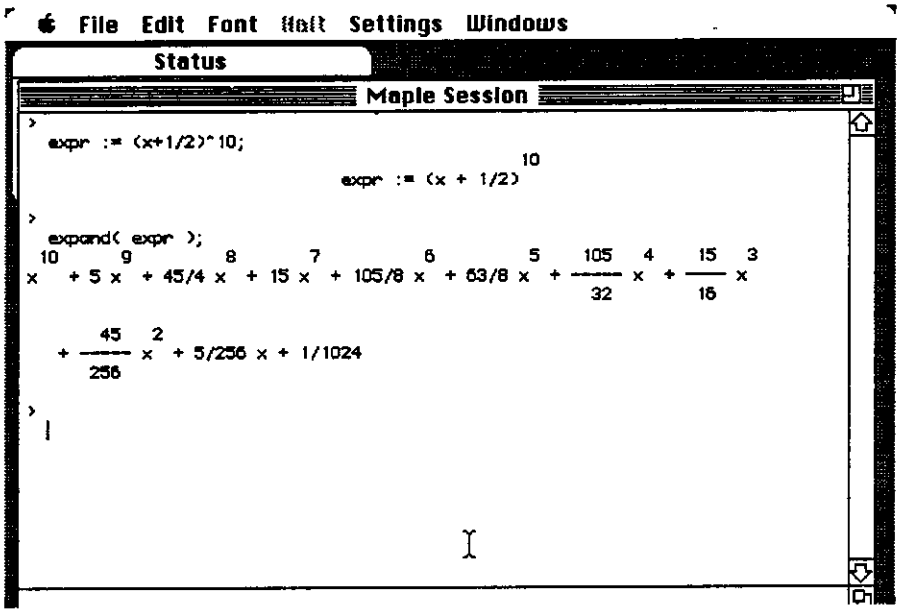

File Edit Font Halt Settings Windows
Status
Maple Session
> expr := (x+1/2)^10;
expr := (x + 1/2)^10
> expand(expr);
x^10 + 5 x^9 + 45/4 x^8 + 15 x^7 + 105/8 x^6 + 63/8 x^5 + 105/32 x^4 + 15/16 x^3
+ 45/256 x^2 + 5/256 x + 1/1024
>

FIGURE 7. Wide expression fitting across a window

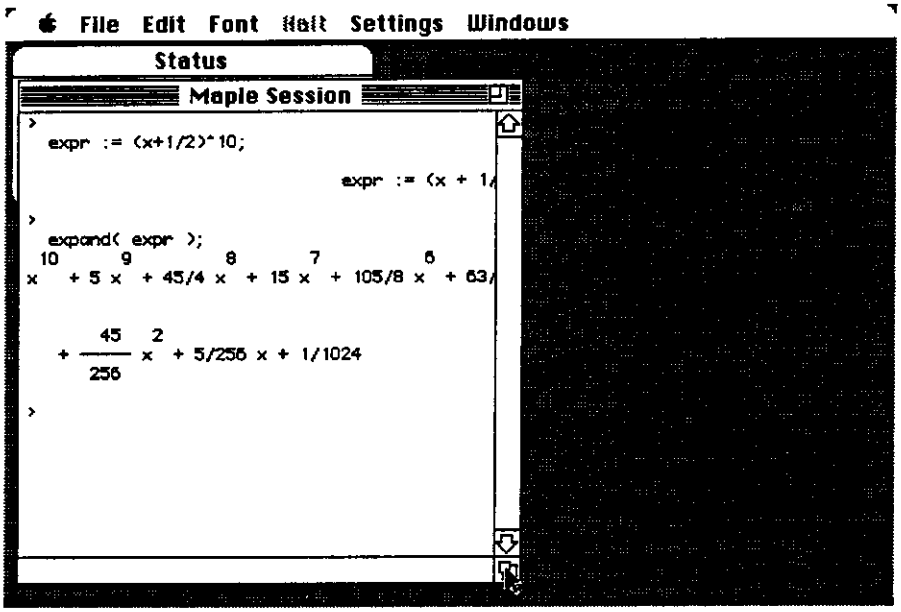

File Edit Font Halt Settings Windows
Status
Maple Session
> expr := (x+1/2)^10;
expr := (x + 1
> expand(expr);
x^10 + 5 x^9 + 45/4 x^8 + 15 x^7 + 105/8 x^6 + 63
+ 45/256 x^2 + 5/256 x + 1/1024
>

FIGURE 8. Wide expression now partially out of view

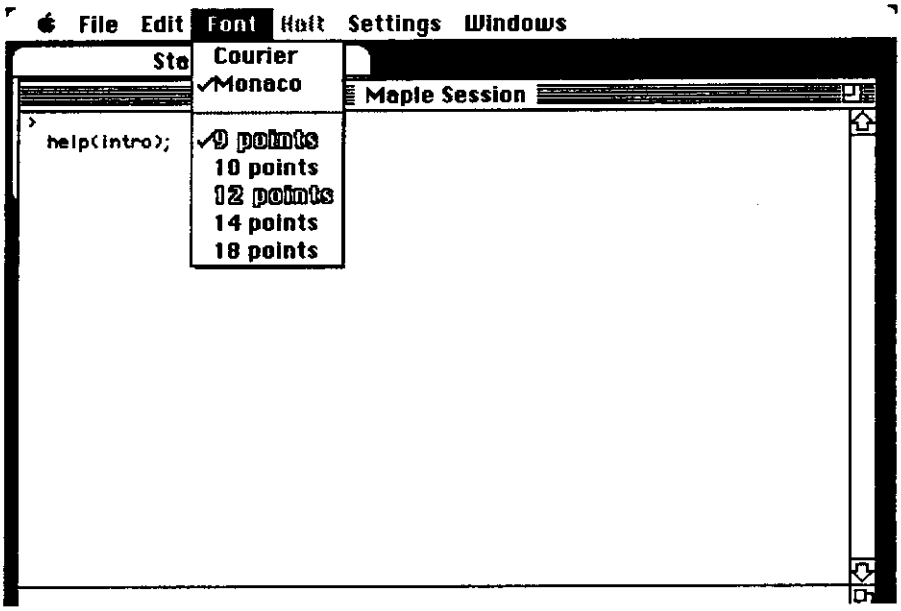


FIGURE 9. The **Font** menu

font sizes or the size of the **Maple Session** window, Maple will adjust the formatting of subsequent expressions so that they fit well across the window.

2.6 Evaluating expressions using enter

What happens when you press the enter key? When you edited `10!` and placed an extra `0` after the `1` and then pressed enter, how did Maple know to evaluate the entire expression instead of evaluating the expression up to the insertion bar? When the enter key is pressed, Maple reads for evaluation whatever the selected or highlighted expression may be. This may be a single character (in which case it's not likely to be a completely terminated Maple expression), or it may be several pages of commands, each command being terminated with a semicolon. As long as there is a selected expression at the time that the enter key is pressed, Maple will evaluate the expression.

If there is no selected expression, then Maple evaluates whichever line the insertion bar is on. Maple uses the entire line, not just the portion before the insertion bar. The simplest way of using Maple is to type an expression on a line (terminated by a semicolon, of course), press enter, review the output or result of the computation, and then type another expression when the prompt appears. Since the insertion bar will be on the line containing an expression when these one-line expressions are entered, this procedure conveniently causes the entire expression to be evaluated.

Unless you do extensive editing of previous expressions and move your insertion bar about, it's not likely that you will type an expression to be evaluated on the same line as the prompt, since Maple always drops down to the next line after having displayed the prompt. If you do type something onto a line beginning with a prompt, you can recover by inserting a [return] between the prompt and the expression.

If you accidentally type an expression on the same line as the prompt (>), you risk having Maple read the entire line as an expression or part of an expression starting with a "greater than" comparison. Consequently, be sure to type expressions to be evaluated on a different line from the one containing the prompt.

2.7 Computations

2.7.1 Progress cues

As you enter an expression for evaluation, the cursor changes from an I-beam to a watch, as mentioned in Section 2.1. If the computation is very quick, the change from an I-beam to a watch and then back again to the I-beam may be so fast that you don't notice. If the computation involves many calculations and takes a while, the watch is a visible cue that Maple is still evaluating the expression.

How can you tell that Maple actually is making some progress and has not simply displayed the watch cursor and then halted? The watch hand will spin to show the march of time and the march of your computation's progress. The watch hand advances a bit each time a Maple statement is evaluated. Since most Maple commands call upon Maple routines consisting of many statements, the watch hand will march forward, although not always at a constant speed, since some Maple statements take longer to compute than others. Some Maple calculations are completed within one statement but take a while to compute. Arithmetic involving very large numbers (e.g., large factorial calculations) are an example. These calculations may cause the watch hand to appear frozen temporarily, but time will march on.

During a Maple session, many intermediate expressions are generated, not all of which are needed permanently. Therefore, it's crucial that unreferenced expressions be collected from time to time so that otherwise wasted memory space can be reused. Occasionally, you will see the cursor turn into a garbage can for a brief moment. This is called *garbage collection*. See Section 5.14 for more information on garbage collection.

2.7.2 Stopping a computation

As soon as a Maple computation begins, the **Halt** menu, which is ordinarily dimmed, appears as a normal enabled menu. The name **Halt** will turn from its normal gray color to black. This shows that there is a computation in progress and that the **Halt** menu can be used to force the computation to halt before finishing normally. Of course, if you force the premature termination of a computation, you will not expect to see its normal result.

There are two possibilities for termination (see Figure 10):

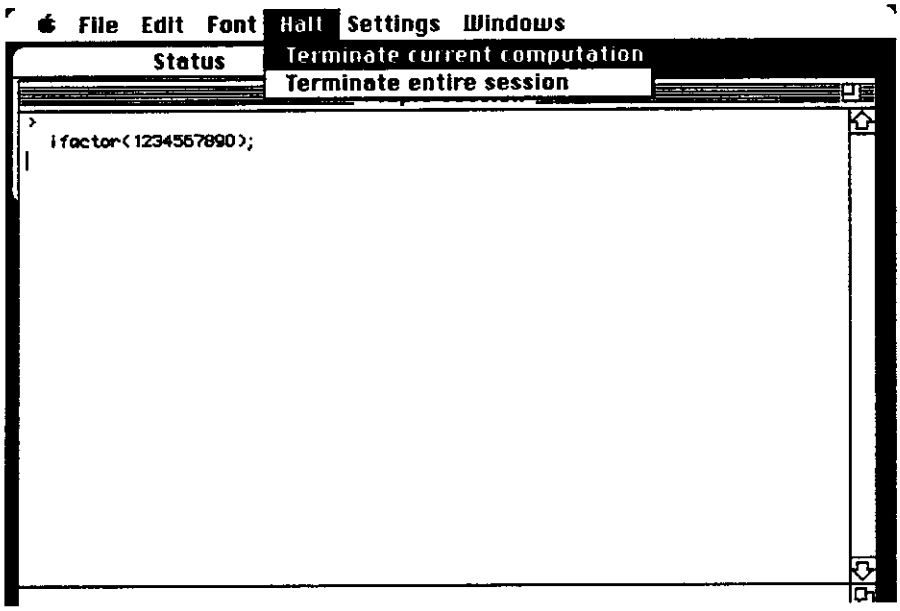


FIGURE 10. The **Halt** menu

1. **Terminate current computation**: This action interrupts the current computation but causes the session to resume.

2. **Terminate entire session**: This action forces the entire session to be terminated and the Maple program to quit immediately.

To safely resume from a computation in progress, especially when memory allocation and garbage collection may be involved, Maple will halt only at the end of a Maple statement. Since most single Maple statements are executed very quickly, normally you will not notice any delay between selecting the **Terminate current computation** item and seeing the computation stopped. Occasionally, if you happen to interrupt a large internal calculation such as arithmetic with very large integers, the computation may terminate only after a noticeable delay. (The **Terminate entire session** item takes no care necessary to resume the session, so its effect is always immediate.)

CHAPTER 3

Computing with Maple

3.1 Maple variables

You can label a calculated result for further use, as was done with `expr1 := x-1;` in Example 1.

variable := *expression* ;

is the general form of the action called *assigning a Maple variable a value*. We use the terms *variable* and *label for a result* interchangeably. However, unlike many conventional programming languages, Maple can use variables in the sense of "a mathematical unknown" as well.

A Maple *programming variable* is one that is used for labeling a result, as was done with *expr1* in Example 1. A variable with no assigned value is regarded as a *mathematical variable* or *unassigned variable*. These are used as "algebraic unknowns," as with x and y in Example 1. If you don't wish to sort out which Maple variables have assigned values and which ones don't, then you can consider Maple mathematical variables as having their own name as their assigned value.

A Maple variable ordinarily starts with a letter and is followed by up to 498 letters, digits, or underscores. Of course, typical variable names aren't close to the maximum length; they are convenient to type and to remember. Weird (but just as valid) Maple names can be created by surrounding characters in backquotes (`), also called "left quotes". Names in backquotes can include any character from the keyboard (including blanks), not just letters, digits, and underscores. These are a few examples of valid Maple names:

```
expr1      x                  T                  Formula_47
d2DYDX     `a funny name? `   ` The answer is= `  `SessionLog.m`
```

● ATTENTION

1. Maple distinguishes between lowercase and uppercase letters. For example, on most computers the name `xyz` is different from `xyZ`.

2. The backquotes mark the boundaries of the name but are not part of it. Thus, Maple considers `x` to mean the same thing as x. However, the name `1` is different from 1. The latter is a number, not a name, and it can't be assigned a value or stand for an algebraic unknown. The former can be used either as a programming or mathematical variable, although it would make life very confusing for anyone trying to read printed or displayed results!

3. The meaning of the mark ` is entirely different from that of the mark ', which is an apostrophe, or "right quote" (see Section 4.6). Be careful to use ` when you mean to!

4. Actually, it's possible to start a variable name with an underscore character (_) instead of a letter. However, Maple's internal programming uses such "underscore variables." To avoid being disappointed or overruled, you should refrain from using such names yourself.

Example 6

Assignment of values (labeling)

```
#"force" is a label for a famous formula.
>
   force := mass*acceleration;
                              force := mass acceleration

#We now assign mass and acceleration particular values.
>
   mass := 3000;
                                    mass := 3000

>
   acceleration := 9.8;
                                 acceleration := 9.8

#The value of "force" below is no longer a formula, but
#incorporates the assigned values of m and c.
>
   force;
                                       29400.0

#Another example, showing one use for a variable that includes
#blank spaces in its name.
>
   distance1 := 10;
                                   distance1 := 10

>
   distance2 := 50;
                                   distance2 := 50

>
   `Total energy` := force *(distance2 - distance1);
                             Total energy := 1176000.0
```

3.2 Printing values: `print` and `lprint`

When you enter an expression or assignment into Maple, the result of evaluation is printed out by default. (As we have mentioned previously, ending a command with a colon instead of a semicolon will prevent the printing of the result.) `print` and `lprint` are commands for displaying the values of one or more expressions at once. A command of the form

`lprint`(*expr1*, *expr2*, *expr3*, ...)

prints out the values that result from evaluating *expr1*, *expr2*, and so on. "lprint" stands for "linear print"—that is, the expression is printed in a linear format, as opposed to the two-dimensional format used in prettyprinting. Thus, the values will all appear on one line (subject to the width of your session window for long answers). The command

`print`(*expr1*, *expr2*, *expr3*, ...)

"prettyprints" the values, separated by commas. For both `lprint` and `print`, variable names surrounded by left quotes (`) will be printed without them.

Example 7
`print` and `lprint`

```
> a := 3/47;
```

$$a := \frac{3}{47}$$

```
> b := (a+x)^2/x;
```

$$b := \frac{(3/47 + x)^2}{x}$$

```
> lprint(a,b,a-1);
3/47   (3/47+x)**2/x   -44/47
> print(a,b,a-1);
```

$$3/47, \frac{(3/47 + x)^2}{x}, -\frac{44}{47}$$

```
> lprint(`a's value is`,a);
a's value is   3/47
> print(`a's value is`,a);
                          a's value is, 3/47
```

3.3 Built-in commands for mathematical computation

Maple "knows" about certain mathematical objects—numbers, symbolic constants (π, e, etc.), multivariate polynomials and rational functions, and expressions involving standard mathematical functions. An obvious way to see this is to notice that Maple automatically *simplifies* expressions: $\cos 2\pi$ to 1, or $e^{2 \log \sqrt{2}}$ to 2. Tables 2 and 3 show some of the mathematical constants and functions that Maple knows about. Example 8 shows some of Maple's automatic mathematical simplifications.

TABLE 2
Mathematical constants known to Maple

Constant	Maple name
integers	`1, 2, ...`
rational numbers	`3/5, -1/3, ...`
floating-point numbers	`1.0, .002, Float(35,-47), ...`
true, false	`true, false`
π	`Pi`
e (natural log base)	`exp(1), E`
$\sqrt{-1}$	`I`
Euler's constant γ	`gamma`

See MRM or **`help(inames);`** for a comprehensive list of constants and for further details.

● ATTENTION

Maple spells π with a capital "P" and a lowercase "i." `PI` and `pi` are not substitutes for `Pi`! Similarly, use `I` and not `i` for the imaginary number.

Example 8
Assortment of automatic mathematical simplifications

```
> 2*abs(sin(-Pi/6));
                                1

> GAMMA(7)-6!;
                                0

> exp(2*ln(x));
                                 2
                                x
```

TABLE 3
Mathematical functions known to Maple

Function	Maple name
exponential e^x	`exp(x)`
natural logarithm $\ln x$*	`log(x)` or `ln(x)`
square root	`sqrt`
absolute value	`abs`
min, max	`min, max`
greatest common divisor	`gcd`
least common multiple	`lcm`
modular arithmetic $a \bmod b$	`a mod b` or `` `mod`(a,b) ``
positive representation	`modp(a,b)`
symmetric representation	`mods(a,b)`
gcd of a and b mod p	`Gcd(a,b) mod p`
sine, cosine, tangent†	`sin, cos, tan`
secant, cosecant, cotangent	`sec, csc, cot`
inverse trigonometric functions	`arcsin, arccos, arctan, arcsec, arccsc, arccot`
hyperbolic trigonometric functions	`sinh, cosh, tanh, sech, csch, coth`
inverse hyperbolic functions	`arcsinh, arccosh, arctanh arcsech, arccsch, arccoth`
$n!$, $\Gamma(x)$ (generalized factorial)	`n!, GAMMA(x)`
$\binom{n}{m}$ (binomial coefficient)	`binomial(n,m)`
n-th Bernoulli number	`bernoulli(n)`
n-th Euler number	`euler(n)`
Stirling numbers	`stirl(n,m), stir2(n,m)`
derivatives of $\ln \Gamma$ $\psi(x), \psi'(x) \ldots \psi^{(n)}(x) \ldots$	`Psi(x), Psi(n,x)`
harmonic numbers	`H(x)`
Riemann ζ function (and derivatives)	`Zeta(x), Zeta(n,x)`
error function $\int_0^x 2e^{-t^2}/\sqrt{\pi}\, dt$	`erf(x)`
dilogarithm function $\int_1^x \ln t/(1-t)\, dt$	`dilog(x)`
Bessel functions of the first kind $J_v(x)$	`bessel(v,x)`
exponential/trigonometric integrals	`Ei(x), Ci(x), Si(x)`
Fresnel integrals	`C(x), S(x)`
hypergeometric functions	`hypergeom([],[],z)`

See MRM or **`help(intfcns)`** for a comprehensive list of functions, and for further details. You may also wish to consult standard reference works on mathematical functions such as *Handbook of Mathematical Functions*, M. Abramowitz and I. Stegun, eds.

*$\log_{10} x$ can be computed through `ln(x)/ln(10)`.

†Trigonometric functions use radians, not degrees.

Maple's built-in knowledge extends to performing numerical evaluation or approximation, differentiation, and integration, to solving equations, and to finding limits involving mathematical expressions. You can tap this knowledge by giving commands to Maple to perform the desired operation—`diff` to differentiate, `solve` to solve equations, and so on. The next section examines some of Maple's commands for mathematical calculation.

You may have noticed that in Example 8, Maple handled the expression $e^{2 \ln x}$, even though x had no assigned value. Maple is designed to handle mathematical operations involving symbols (variables with no assigned values) as well as numbers.

Example 9
Maple calculations with mathematical unknowns

```
#Maple expressions can involve symbolic unknowns as well as numbers.
#Such expressions can be labeled, just as with numerical results.
#Maple automatically performs a limited amount of simplification
#as it evaluates the expression.
>
   expr1 := (x + y + x + x*x*x + x)/2;
                                                    3
                        expr1 := 3/2 x + 1/2 y + 1/2 x

>
   expr2 := ((4*expr1) -2*y)/x;
                                                  3
                                          6 x + 2 x
                               expr2 := ------------
                                            x

#Symbolic expressions can include any function.
>
   bigexpr :=  sin( expr1 ) / ln ( expr2 ^2 );
                                                        3
                                          sin(3/2 x + 1/2 y + 1/2 x )
                        bigexpr := ---------------------------------
                                                  3 2
                                              (6 x + 2 x )
                                   ln(-----------------)
                                                 2
                                               x
```

While expressions involving only numbers are always fully reduced to the simplest fraction, Example 9 shows that Maple's automatic procedures for reducing and simplifying symbolic expressions do not always reduce things to the desired form. In the next section, we learn about symbolic calculations Maple can perform that go far beyond simple arithmetic. In Sections 4.3 and 4.4, we return to the problem of how to simplify a symbolic answer into the form you wish for it.

3.4 Introducing Maple's mathematical commands

Maple has preprogrammed knowledge of common mathematical operations beyond its knowledge of arithmetic and simplification. When the operation involves only

one expression, you can tap Maple's calculational power by entering a command of the form *operation*(*expression*);—for example, `abs(-55);`. Sometimes the operation involves two or more expressions. Thus, you could compute the greatest common divisor of 55 and 22 with `gcd(55,22);`. In addition to mathematical functions such as `gcd` or `abs`, Maple has a number of "computer functions" that have a similar form: they take one or more expressions as *arguments* to the function and return a single object (number, expression, etc.) as their *value* or *result*. For example, Maple has a function **solve**. The result of evaluating `solve(3*x-5=x, x);` is $5/2$—what you get when you solve the equation for x.

In this section, we take a brief tour of some of the more commonly used computational functions, including those for solving equations, doing calculus, and plotting graphs. We will also take a look at **subs**, a useful function for evaluating expressions at a point and for applying identities to expressions. (Maple's facilities for matrix and vector linear algebra are discussed in detail in Chapter 4.)

3.4.1 Differentiation

diff(*expr*, *variable of differentiation*)

diff computes the derivative of *expr* with respect to a variable.

Example 10
First derivative with respect to x via `diff`

```
> diff( sin(ln(x^2 + 1) / 3), x);
```

$$2/3\,\frac{\cos(1/3\,\ln(x^2+1))\,x}{x^2+1}$$

Higher-order partial derivatives can be taken by `diff` with a sequence of variables. The following example computes the partial derivative $\partial^2/\partial t\partial s$ of an expression in s and t.

Example 11
Partial derivative with respect to s and t

```
#Compute a second-order partial derivative of an expression.
#
> (t^2 -s) / (s^3 -1):
> diff(", t, s);
```

$$-6\,\frac{t\,s^2}{(s^3-1)^2}$$

3.4.2 Indefinite and definite integration

int(*integrand*, *variable of integration*) (indefinite integration)
int(*integrand*, *variable of integration* = *lo* .. *hi*) (definite integration)

Maple will try to calculate a definite or indefinite integral when the **int** command is given. However, unlike differentiation, where the computer coding of "how to differentiate" is straightforward, there is no guaranteed success for integration. When Maple fails to solve an integration problem, the request for the calculation, perhaps simplified somewhat, is left as the symbolic "answer."

Example 12
Symbolic indefinite and definite integration with int

```
#Integrate x^3*cos(x) with respect to x.
#
>
   int(x^3*cos(x), x);
                3                2
              x  sin(x) + 3 x  cos(x) - 6 cos(x) - 6 x sin(x)

#A definite integral.
#
>
   int( (x^3)*sqrt(x^2-a^2), x=a..2*a);
                                      5  1/2
                                 14/5 a  3

#Another indefinite integral.
#
>
   int( (x^2)/sqrt(1-x^2), x);
                                  2 1/2
                    - 1/2 x (1 - x )    + 1/2 arcsin(x)

#Other indefinite integrals.
>
   int( cos(theta)^3, theta=0..Pi/2);
                                   2/3

>
   int( 1/(1+x^2), x=0..infinity);
                                  1/2 Pi

#Sometimes Maple can't compute a closed-form solution.
#
>
   int( x^(-2), x=-1..1);
                                    1
                                    /
                                   |   1
                                   |  ---- dx
                                   |   2
                                  /    x
                                 -1

#The "answer" is just the prettyprinted version of the original command
#expression.
#
>
   lprint(");
int(1/x**2,x = -1 .. 1)
```

3.4.3 Summation

sum(*summand, index variable*) (indefinite summation)
sum(*summand, index variable = lo .. hi*) (definite summation)

Example 13
Summation with sum

```
#The sum of the first 100 integers.
>
   sum( i, i=1..100);
                                  5050

#A definite symbolic sum.
>
   sum( i^2, i=1..n);
                          3              2
                1/3 (n + 1)  - 1/2 (n + 1)  + 1/6 n + 1/6

#Another definite symbolic sum.
>
   sum(2^i, i=m..n);
                                (n + 1)    m
                               2        - 2

#An "indefinite" symbolic sum is similar to an indefinite integral —
#it is the function F such that F(b+1)-F(a) would yield the sum
#with index ranging from a to b.
#
>
   sum(1/(i^2),i);
                                - Psi(1, i)

#As with integration, the request is the returned value when no formula
#can be found.
#
>
   sum(log(i),i=1..n);
                                  n
                                 ---
                                 \
                                  )   ln(i)
                                 /
                                 ---
                                 i = 1

>
   lprint(");
sum(ln(i),i = 1 .. n)
```

3.4.4 Solution of algebraic equations

solve(*equation*)
solve(*equation, variable*) (solve an equation for an unknown)

In its simplest forms, **solve** takes an equation in one unknown for an argument, as in `solve(3*x + 4 = 5*x)`, or an equation in several unknowns with one to solve for, as in `solve(ln(x^2-1) = a, x)`. The value of `solve` will be the number or expression that is the solution of the equation.

If `solve` finds more than one solution, you will see the solutions separated by commas. In the case of multiple solutions, `solve` returns an *expression sequence*. Expression sequences are discussed in Section 4.10. By assigning the result of `solve` to a programming variable, (say, `s`) you can refer to a particular component of a multiple solution using a *selection operation* or *indexing* notation (e.g., `s[1]`, `s[2]`, etc.). Further discussion of how to get at components of structured expressions can be found in Sections 4.9 and 4.10.2.

Example 14
Solution of one equation and one unknown

```
#Solution of a single equation with one unknown.
#
>
   solve( 3*x + 4 = 5*x);
                                    2

#Solution of a single equation with more than one solution.
#
>
   solve(ln(x^2 - 1) = a,x);
                                  1/2                    1/2
              - 1/2 (4 exp(a) + 4)   , 1/2 (4 exp(a) + 4)

#When an expression is given instead of an equation, an "=0" is
#implicitly assumed.
#
>
   solve( q^3-k, q);
        1/3         1/3        1/2  1/3          1/3        1/2  1/3
       k   , - 1/2 k    + 1/2 3    k    I, - 1/2 k    - 1/2 3    k    I

#One can assign the result of solve to a programming variable, and then
#refer to each solution by an index.
#
>
   solutions:=";
             1/3         1/3        1/2  1/3          1/3        1/2  1/3
 solutions := k   , - 1/2 k    + 1/2 3    k    I, - 1/2 k    - 1/2 3    k    I

>
   solutions[1];
                                    1/3
                                   k

>
   solutions[3];
                              1/3        1/2  1/3
                        - 1/2 k    - 1/2 3    k    I
```

solve(*set of equations*, *set of variables*)
(solve a set of equations for a set of unknowns)

If you give `solve` a set of equations and a set of unknowns to solve for, it will return a set of solutions (or several sets, if there is more than one solution). For example,

```
solve({a = b + 2, b - a = t}, {a, b})
```

is a command to solve the system of two equations for the unknowns a and b.

Each element of the solution set will be an equation of the form *var* = *expr*, where *var* is one of the variables being solved for.[1] When a variable being solved for can take on any value in the solution, the solution set includes a "tautological equation" (e.g., `x = x`).

Example 15
Solution of system of equations

```
#Solution of a set of linear equations.  The variables to be solved for
#are also collected together into a set.  A solution set is returned.
#
>
   eqnset := { x+y=b,  a*x-2/3*y = k};
                    eqnset := {x + y = b, a x - 2/3 y = k}

>
   varset := {x,y};
                                varset := {x, y}

>
   SolutionSet := solve(eqnset, varset);
                                                  a b - k          3 k + 2 b
              SolutionSet := {y = 3 ----------, x = ----------}
                                                  3 a + 2          3 a + 2

#The assign function will take the solution set from solve and assign
#the specified variables accordingly.
#
>
   assign(SolutionSet);
>
   x; y;
                                  3 k + 2 b
                                  ---------
                                  3 a + 2

                                    a b - k
                                3 ----------
                                    3 a + 2

#The values of the equations now include the updated information
#about x and y.
#
>
   eqnset;
        3 k + 2 b       a b - k       a (3 k + 2 b)       a b - k
    {   ---------  + 3 ----------  =  b,  -------------  - 2 ----------  = k}
        3 a + 2         3 a + 2          3 a + 2           3 a + 2

>
   simplify(eqnset);
                                {b = b, k = k}
```

[1] Sets and equations as data structures are explained in more detail in Section 4.9.

```
#Unassign the names x and y so that their values will again become
#mathematical unknowns (no assigned value). ``Unassignment'' is
#discussed fully in Section 4.6.
#
>
   x := 'x';  y := 'y';
                                    x := x

                                    y := y

#The equations always reflect this most current information about
#x and y.
>
   eqnset;
                          {x + y = b, a x - 2/3 y = k}

#Solution of a nonlinear system of polynomials with two solutions.
>
   solve({y^2+1=x, x+2=y}, {x,y});
                          2                                  2
          {y = RootOf(_Z  + 3 - _Z), x = - 2 + RootOf(_Z  + 3 - _Z)}
```

Maple will always find the solution to a system of linear equations should the solution exist (given enough computer time and memory for computing). However, its methods for nonlinear equations do not always succeed even when a solution exists. If it cannot find any solutions, it will print out **Warning: no solutions found.** Even if Maple finds some solutions to an equation or system of equations, the programming may not direct it to find *all* of them.

Example 16

What happens when no solutions, or only some solutions, are found

```
#A message is printed when no solution can be found.
#
>
   solve(x=x+1,x);
                          Warning: no solutions found

>
   solve({cos(x*Pi)=0,cot(x*Pi/3)=0},{x});
                          Warning: no solutions found

#Maple does not always give all solutions.
#
>
   solve(cos(x*Pi)=0,x);
                                    1/2

#If no variables are supplied, then Maple will pick which unknowns
#to solve for.  Be sure that Maple agrees with you about what
#the unknown is!
#
>
   solve( a*x^2 + b*x + c =0);
                                 2
                {c = - a x  - b x, x = x, a = a, b = b}
```

Sometimes a solution involves *algebraic numbers.*[2] Some algebraic numbers have simple forms: 4 and -4 are the zeroes of $x^2 - 16$; $\sqrt{2}$ is a zero of $z^2 - 2$. However, sometimes the simplest way to refer to an algebraic number is to refer to the specific polynomial equation it is a root of. Maple refers to the three roots of $x^3 + 3x = 2$ (approximately .596 and $-.298 \pm 1.81i$) as

```
RootOf(_Z^3 + 3*_Z - 2).
```

When you see a "RootOf" expression in a result returned by `solve`, it actually is a placeholder for several solutions—one for each possible root of the polynomial described in the "RootOf."[3] In Example 15, the result of

```
solve({y^2 + 1 = x, x+2=y}, {x,y});
```

is

```
                                   2                                  2
{y = RootOf(_Z  + 3 - _Z), x = - 2 + RootOf(_Z  + 3 - _Z)}.
```

This is actually two distinct solutions rolled into one. Since the two roots of $x^2 + 3 - x = 0$ are $(1 \pm \sqrt{11}i)/2$,[4] the two solutions to the system of equations are

$$y = (1 + \sqrt{11}i)/2, \quad x = -2 + (1 + \sqrt{11}i)/2$$

and

$$y = (1 - \sqrt{11}i)/2, \quad x = -2 + (1 - \sqrt{11}i)/2 .$$

Note that when you want to generate explicit multiple solutions from a `solve` solution with a `RootOf` in it, you must be consistent in the choice of roots throughout the solution.

Sometimes it is impossible, or very expensive, to find an exact solution of equations that are known to have a numerical solution. Maple's **`fsolve`** procedure will use numerical methods to determine approximate roots. Consult **`help(fsolve)`** or MRM for further information.

[2] An algebraic number is a real or complex number that satisfies a polynomial equation.

[3] Maple's `simplify` command is useful in reducing the size of solutions that have RootOfs in them. See Section 4.3.1.

[4] It doesn't matter whether we use x or `_Z` as the variable when describing things to ourselves. However, Maple prefers using the variable named `_Z` in its RootOfs.

3.4.5 Other solvers: dsolve, etc.

dsolve can find closed-form and power series solutions to many first-, second-, or higher-order ordinary differential equations. It will also try to solve systems of equations, either explicitly through the method of Laplace transforms or approximately in terms of power series. There are several forms of the command:

dsolve(*equation*, *depvar*(*indvar*));
dsolve(*set of equations and initial conditions*, *set of variables*);
dsolve(*set of equations and initial conditions*, *set of variables*, *option*);

The options that can be used with dsolve are shown in Table 4.

TABLE 4
Options used with dsolve

When *option* is	dsolve (..., *option*); computes
laplace	A solution using the method of Laplace transforms to solve an equation or system of equations.
series	A power-series solution (the number of terms is dictated by the value of the global variable Order, default 6).
explicit	A solution using solve to force the solution to be returned explicitly in terms of the dependent variable.

To solve systems of equations, use laplace or series.

Example 17
Solving differential equations with dsolve

```
#A differential equation is an equation involving an unknown function y
#of an independent variable x.  It is necessary to say "diff(y(x),x)"
#instead of "diff(y,x)" because diff needs explicit declaration of y's
#dependency upon x.
#
>
   deq := diff(y(x),x)*x^2+y=0;
```

$$deq := \left(\frac{d}{dx} y(x)\right) x^2 + y = 0$$

```
#A differential-equation solution is called for by including the
#equation and the names of the dependent and independent variables.
#
>
   dsolve(deq, y(x));
```

$$y(x) = \exp(1/x)\, C$$

```
#An initial-value problem is described by giving a set of equations
#as the first argument.
>
   dsolve({deq, y(1)=a},y(x));
                                        exp(1/x) a
                              y(x) = ---------------
                                        exp(1)

#Part of dsolve's programming includes Kovacic's algorithm for solving
#second-order linear homogeneous equations with rational function
#coefficients.
#
>
   dsolve(x^2*diff(y(x),x,x) + 1/x*diff(y(x),x) = 0,y(x));
                                1                          1/2    1/2          1
      y(x) = C + C1 x exp(1/2 ------) + 1/2 C1 Pi    (-2)    erf(----------)
                                 2                                  1/2
                                x                                  2    x

#dsolve can't solve all differential equations exactly, of course.
#
>
   dsolve(  diff(f(x),x) + f(x)^5*x = sin(x), f(x));
                          Warning: no solutions found.

#But sometimes the first terms of a power-series solution can be
#found.
#
>
   dsolve( {f(0)=1/2, diff(f(x),x) + f(x)^5*x = sin(x)}, f(x), series);
                                  31   2     977    4      6
                      f(x) = 1/2  + --- x  - ------- x  + O(x )
                                  64         12288
```

Maple has a number of other solvers, including **isolve**, for finding integer solutions to algebraic equations; **msolve**, for finding solutions to equations modulo p; **rsolve**, for finding solutions to recurrence relations; and **fsolve**, which uses numerical methods to find approximate roots to algebraic equations. See MRM or **help** for further details.

3.4.6 Expression substitution with subs

Often in a calculation of several steps, a known identity such as $f = m * a$, $v = i * r$, or $x = 5/3$ has to be applied to an expression. We can achieve this by replacing every f in the expression by $m * a$, or every v by $i * r$, and so on. This can be done in Maple with the **subs** command. The result of the command

subs(*var* = *replacement*, *expression*)

is an expression where every occurrence of the symbol *var* in *expression* is replaced by *replacement*. A double substitution can be specified by:

subs(*var1* = *replacement1*, *var2* = *replacement2*, *expression*).

The result is computed by first replacing every occurrence of *var1* by *replacement1*, then replacing every occurrence of *var2* by *replacement2*. Multiple (three,

four, or more) substitutions can be specified by using subs in a similar way—a sequence of variable/replacements, followed by the expression to be operated upon.

subs is not limited to symbol replacement. For example, it can be used to replace one term of a sum with another expression. Knowing precisely what subexpressions can be replaced within an expression involves an understanding of which subexpressions are accessible by the **op** function. See Section 4.9 and MRM.

Example 18

Substitution of an expression for another via subs

```
#The "subs" (substitute) command can be used to take solutions and
# plug them into the original equation to verify the solutions.
#
>
   soln:= solve(3*x-y=x,x);
                                soln := 1/2 y

>
   subs(x=soln, 3*x-y=x);
                                1/2 y = 1/2 y

# Multiple replacement of symbols by other symbols or expressions.
#
>
   msol := solve( e=m*c^2,m);
                                       e
                                msol := ----
                                         2
                                        c

>
   subs(m=msol, a=9.8, c=300000, f = m * a);
                                                 -9
                            f = .1088888889*10    e

# Note that the replacements occur in order from left to right as they are
# specified.  In the (misguided) calculation below, the substitution of
# "c" occurs before there is any "c" in the expression to replace!
>
   subs(c=300000, m=msol, a=9.8, f= m * a);
                                       e
                                f = 9.8 ----
                                         2
                                        c

#The following shows replacement of an expression by another expression.
#Replace tangents by a ratio of sines and cosines, and then replace
#the squared cosines.
#
>
   expr2 := cos(x)^2 + tan(x)^2;
                                        2         2
                           expr2 := cos(x)  + tan(x)

>
   subs(tan(x)=sin(x)/cos(x), cos(x)^2=1-sin(x)^2, expr2);
                                              2
                                   2     sin(x)
                            1 - sin(x)  + ------
                                              2
                                          cos(x)
```

```
# Well, that almost worked. The quotient with cos(x)^2
# is being regarded by Maple as being cos(x)^(-2) in a product.
#
> subs(cos(x)^(-2)=1/(1-sin(x)^2),");
```

$$1 - \sin(x)^2 + \frac{\sin(x)^2}{1 - \sin(x)^2}$$

The first argument to subs can also be a set (or a list) of equations. Thus, you can use the result of solve for substitutions even when the result is a set. However, when a set or list of substitutions is given, all of them are performed simultaneously instead of sequentially.

Example 19
Simultaneous substitution of a set of expressions

```
#Find a cubic polynomial p(x) that satisfies p(1)=1, p'(1)=2, p''(1)=3
#
> p := a*x^2 + b*x + c;
```

$$p := a x^2 + b x + c$$

```
> conditions := subs(x=1, {p=1, diff(p,x)=2, diff(p,x,x)=3});
```

$$conditions := \{2 a = 3,\ a + b + c = 1,\ 2 a + b = 2\}$$

```
> soln:= solve(conditions,{a,b,c});
```

$$soln := \{b = -1,\ c = 1/2,\ a = 3/2\}$$

```
#Display the polynomial by plugging in the substitutions.
#a, b, and c are not assigned values, unlike what would happen with
#the "assign" function.
#
> subs(soln, p);
```

$$3/2\, x^2 - x + 1/2$$

```
#Compare with the "obviously correct" form of the answer:
#
> 3/2*(x-1)^2 + 2*(x-1) + 1;
```

$$3/2\, (x - 1)^2 + 2 x - 1$$

```
> simplify(");
```

$$3/2\, x^2 - x + 1/2$$

3.4.7 Taylor and asymptotic series

taylor(*expression*, *variable* = *expansion point*, *order of truncation*)
asympt(*expression*, *variable*, *order of truncation*)

Maple can construct any finite number of terms of the Taylor series (more generally, Laurent series) or asymptotic (at infinity) series expansion of an expression. Each function appearing in the *expression* must be known to Maple's **taylor** function either directly, via a procedure for computing its Taylor (Laurent) series coefficients, or indirectly, via a procedure for computing its derivatives using **diff** (symbolic differentiation). `O(x^n)` is tacked on to the end of the series if the terms up to x^{n-1} are not the complete expression. As a rule of thumb, the `O(x^n)` usually appears except when the *expression* is a polynomial of degree less than n in the series variable.

The value of **Order** controls the order of truncation (i.e., the value n above) used during the calculation. The value of `Order` is set initially to 6, but you can change it by assignment, just as you would any other variable.

Example 20

Taylor and asymptotic series calculations

```
#Compute the first few terms of the Taylor series for sin(x) expanded
#at x=0.  The default is to compute up to O(x^6).
#
>
   t1 := taylor( sin(x), x=0);
                                        3          5      6
                      t1 := x - 1/6 x  + 1/120 x  + O(x )

#An optional third parameter specifies the order to override the default
#of 6.
#
>
   t2 := taylor( ln(x), x=1, 4);
                                    2             3           4
            t2 := x - 1 - 1/2 (x - 1)  + 1/3 (x - 1)  + O((x - 1) )

#The order information in series results is used when doing arithmetic
#on series themselves.
#
#In the example below, the result is only O(x^7) since
#t1 is only O(x^6).
#
>
   taylor( t1*t1, x=0, 10);
                           2      4        6      7
                          x  - 1/3 x  + 2/45 x  + O(x )

#Polynomials have series expansions that are exactly correct if the
#order of expansion is larger than the degree of the polynomial.
#
>
   taylor( x^10 + 1, x=0, 20);
                                        10
                                     1 + x

>
   taylor( "*", x=0, 25);
                                     10    20
                                1 + 2 x   + x

>
   taylor(x^10+1,x=0,5);
                                        10
                                   1 + O(x  )
```

```
>
   taylor("*",x=0,25);
                                                10
                                      1 + O(x  )

#Asymptotic series at x=infinity (default order O(x^(-6)).
#
>
   asympt ( exp(1/x)/(1+x^3), x);
                          1     1          1        1
                        ---- + ---- + 1/2 ---- + O(----)
                         3     4          5        6
                        x     x          x        x

#"taylor" can handle more general series as well
#
>
   taylor(exp(sqrt(2*x^3))*log(x),x=0);
     1/2       1/2     1/2   3/2         1/2   3        1/2     1/2   9/2      6
2 ln(x   ) + 2 2    ln(x   ) x    + 2 ln(x   ) x  + 2/3 2    ln(x   ) x    + O(x )
```

3.4.8 Limits of real and complex functions

limit(*expr*, *x* = *a*); (real bidirectional limit)
limit(*expr*, *x* = *a*, *direction*); (directional limit)

The first form of **limit** computes the limiting value of *expr*, an expression in the variable *x*, as *x* approaches the number *a*. That is, `limit` returns *b* if $b = \lim_{x \to a} expr$, in the "real bidirectional" (standard calculus) sense.

As with integration, the programming of `limit` does not always succeed in finding the correct answer. The same expression that was entered is returned when `limit` fails to find an answer.

`limit` assumes that any symbolic unknowns in the expression (other than the variable that is being taken to the limit) are *real* and *nonzero*. Thus, `limit(c^2 * x, x = infinity);` will return `infinity`, but `limit(c * x, x = infinity);` will fail, since the result depends on whether `c` is positive or negative.

Example 21
Real bidirectional limits

```
#An assortment of real bidirectional limit problems
#
>
   r := (x ^2 -1 ) / (11*x^2 - 2*x -9):
>
   limit(sqrt(r), x = 1 );
                                          1/2
                                        2
                                1/2 -------
                                        1/2
                                       5

>
   limit(sin(x)/x, x=0);
                                        1

>
   limit(sin(1/x^2), x=0);
                                    -1 .. 1
```

```
> limit(sin(1/x)/x, x=0);
                                undefined

#Deriving the definite integral of sin from 0 to 1 the hard way —
#taking the limit of the area of a sum of rectangles.
#
> intsin01 := sum(sin(i/n)*1/n,i=1..n);
```

$$\text{intsin01} := -1/2\,\frac{\sin(\frac{n+1}{n})}{n} + 1/2\,\frac{\sin(1/n)\cos(\frac{n+1}{n})}{n\,(\cos(1/n)-1)} + 1/2\,\frac{\sin(1/n)}{n} - 1/2\,\frac{\sin(1/n)\cos(1/n)}{n\,(\cos(1/n)-1)}$$

```
> limit(intsin01,n=infinity);
                                - cos(1) + 1
```

The limit point a can be any real number, an expression, `infinity`, or `-infinity`. If the limit point is `infinity` or `-infinity`, a directional limit is taken instead of a bidirectional one. The result of `limit` may be a number, an expression, `infinity`, `-infinity`, `undefined`, or a range *c..d*. The latter occurs when *expr* has no limiting value at a, or an unknown one, but does have a limiting *range*; that is, all of *expr*'s values lie in the interval $[c, d]$ as x approaches a.

The second form of limit computes directional limits. Table 5 explains the options. The **complex** option computes the multidirectional limit. A result of `infinity` in this case does not mean real positive infinity, but unbounded complex values as the limit point is approached.

TABLE 5
Taking directional limits

When *direction* is	`limit` (*expr*, *x*=*a*, *direction*); computes
left	$\lim_{x \to a^-} expr$.
right	$\lim_{x \to a^+} expr$.
complex	A value of `infinity` here means unbounded complex values as the limit point is approached.
real	The same result as the real bidirectional limit (i.e., when no *direction* is provided), except when the limit point is infinity. If the result of `limit` is b in this situation, then b is the bidirectional limit at infinity; i.e., $b = \lim_{x \to -\infty} expr = \lim_{x \to +\infty} expr$.

Example 22
Other directional limits

```
> limit(tan(x), x=Pi/2, right);
                                  - infinity
#One-sided limit approaching positive infinity from the left.
> limit(x^3, x=infinity);
                                   infinity
#Bidirectional limit.
> limit(tan(x), x = Pi/2, real);
                                  undefined
#A complex multidirectional limit.
> limit(sin(x)/x^2, x=0);
                                  undefined

> limit(sin(x)/x^2, x=0, real);
                                  undefined

> limit(sin(x)/x^2, x=0, complex);
                                   infinity
```

3.4.9 Numerical calculations to any desired precision

evalf(*expr*)
evalf(*expr*, *digits*)

Besides exact integers and fractions, Maple also has *calculator numbers*, known also as *floating-point* numbers. Floating-point arithmetic in Maple simulates a decimal calculator with 10 digits of accuracy by default. Numbers are printed in conventional scientific notation, as a decimal fraction times a power of 10. (A more natural fixed-point notation is used if the number is within a few orders of magnitude of 1.) Maple will convert mixed expressions involving both floating-point and rational numbers into floating-point representation.

Maple differs from ordinary calculators in that you can *increase* or *decrease* the number of digits used in the floating-point arithmetic. The value of the variable **Digits** controls the number of (decimal) digits used during the calculation. It is initially 10, but you can change `Digits` by assigning it a different value, just as you would any other Maple variable. For example, if you wanted a 40-digit calculator, you'd assign `Digits` the value 40 before starting your calculations.

Example 23
Floating-point numbers in Maple

```
#Floating-point instead of exact arithmetic is used when an
#expression contains a floating-point and rational number.
> 1.0 + 3/5;
                                 1.600000000
```

```
> 3.0*10^20 - 2.99*10^13;
                                                21
                                  .2999999701*10

#Floating-point calculations are not always exact.
> 1.0/3.0 + 1.0/3.0 + 1.0/3.0;
                                    .9999999999

> " - 1;
                                                -9
                                     -.1*10

#However, approximation can be a way of conveying useful information that a
#similar rational calculation can obscure.
#
> sum(1/(2*i-1),i=1..10);
                                      31037876
                                     ----------
                                      14549535

> sum(1.0/(2*i-1),i=1..10);
                                    2.133255530

#The value of Digits controls the accuracy of floating-point calculation.
#
> Digits;
                                        10

> Digits := 5:
> 2.0/3.0 + 5;
                                      5.6667

> Digits := 40:
> 2.0**(1/3);
                  1.2599210498948731647672106072782283505 70
```

Floating-point numbers are different from rational numbers in Maple because arithmetic with them is carried out only to a certain point; no attempt is made to keep enough digits to represent the answer exactly. The amount of work needed for a floating-point calculation can be considerably less than for its exact rational counterpart. Thus, floating-point numbers are useful in situations where approximation is good enough to handle the problem. You may notice that Maple floating-point arithmetic typically runs more slowly than the fixed-accuracy floating-point ("real") arithmetic of languages such as Pascal, Fortran, Lisp, or Prolog. This is due in part to the "adjustable accuracy" feature of Maple's arithmetic.

evalf(*expr*) ("evaluate in a floating-point context") will invoke procedures from the Maple library to approximate the numeric expression *expr*. *expr* may include the operations, functions, and constants shown in Tables 1 through 3 (in addition to a few others described in MRM).

Maple users can extend Maple's built-in knowledge of floating-point approximation. For example, defining the Maple procedure `` `evalf/func` `` causes the procedure

to be invoked when `evalf` is applied to an expression involving the function `func`. (See MRM for further details. See also Chapter 5 for how to define a Maple procedure.)

Example 24
Approximating exact results with `evalf`

```
#Evalf is required to force floating-point arithmetic whenever
#functions or expressions involving all rational numbers are involved.
>
   expr1 := tan(1.0) + tan(2);
                        expr1 := 1.557407725 + tan(2)

>
   evalf(expr1);
                                  -.627632138

>
   Digits := 20:
>
   evalf(expr1);
                              -.6276321382615189916

#You can temporarily override the value of Digits for a particular
#evalf computation by including the number of digits as a second
#argument to evalf.
#
>
   evalf(sqrt(2),40);
                  1.4142135623730950488016887242096980785 70

#evalf of a RootOf gives an approximation to an arbitrarily chosen
#real root of the polynomial, if there is one.
#
>
   {solve((x^3+1)*(x^6+3*x+1),x)};
                          1/2                1/2          6
       {-1, 1/2 + 1/2 I 3   , 1/2 - 1/2 I 3   , RootOf(_Z  + 3 _Z + 1)}

>
   evalf(");
{.50000000000000000000 + .86602540378443864675 I,
   .50000000000000000000 - .86602540378443864675 I, -1., -.33379438854319331629
   }
```

3.5 Other functions

Maple has hundreds of built-in mathematical functions besides the ones we've described. Some are useful to many people, such as the functions **numer**(*expr*) and **denom**(*expr*), which give the numerator and denominator of a symbolic quotient of expressions. Others are likely to be useful to specialists in certain fields. All of them, however, obey the format of having arguments and returning a value. Typically, the functions useful to smaller subcommunities of Maple users must be retrieved using the **readlib** command (discussed in Section 3.12) or are in packages.[5]

[5] Maple also has "computer functions" for data structure manipulation. These will be discussed in Chapter 4. Packages and the `with` command are explained in Section 4.14.

MRM provides an explanation of all Maple functions, and some are discussed subsequently in this tutorial. Additionally, descriptions of them are available via `help` (see Section 3.13).

3.6 Graphing and plotting functions on screen and on paper

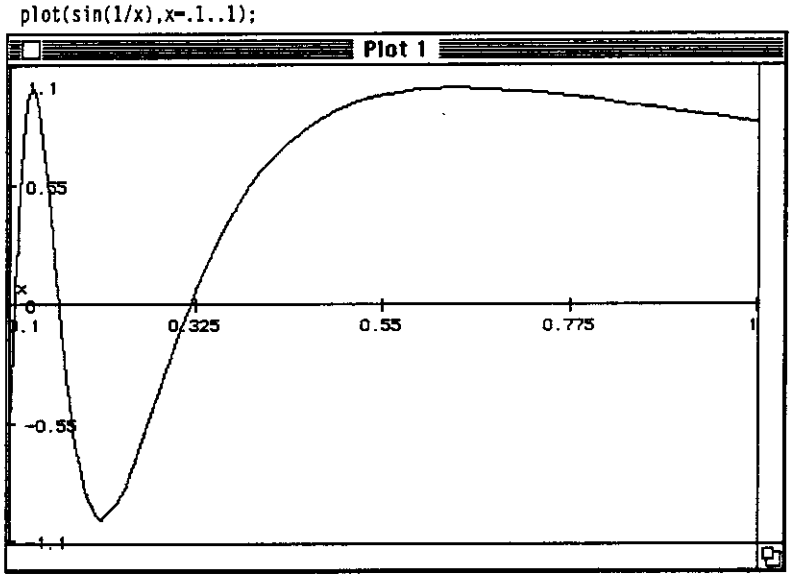
plot(*expr*, *var* = *range*); (plot of a single function)

A real-valued function can be plotted on any interval by means of the **plot** command. `plot` will produce a graph of the expression *expr*, a function of the variable *var* on the specified *range*.

Maple produces plots that appear in a separate resizable plot window. When this plot window is the active window, the results can be saved to a file or printed to a printer. The plots are in PICT format and may be imported directly into other Macintosh applications for further modification.

Example 25
Plot of a single function

```
> plot(sin(1/x),x=.1..1);
```



To get Maple to plot several functions on the same graph, you can give the command

plot(*set of expressions*, *var* = *range*);

where each expression in the *set of expressions* is a real-valued function of the variable *var*.

Maple can also produce parameter plots, plot data points, plot in polar coordinates, and can display a graph within a limited vertical range of values. By default, Maple computes the value of *expr* at a modest number of equally spaced points in the interval. It then inspects the fluctuations in values between points and will select more points within subintervals where there is a great deal of fluctuation. By default, it will then draw the data points connected by specially designed smooth curves (natural cubic splines).[6] The default number of points, as well as the default method of interpolation, can be overridden. See MRM or consult `help(plot);` for further details.

The `plot` command supports plots on several other kinds of graphics displays. You should set the global variable **`plotdevice`** to one of the values listed in Table 6 if you wish to plot using the special capabilities of one of the corresponding displays.

TABLE 6
`plotdevice` values

Value	Purpose
`mac` (default)	Plot will appear in a separate window on the Macintosh screen.
`char`	Character plots (for terminals with no graphics support).
`vt100`	Like character plots, but will use horizontal line graphics characters of VT100. Especially effective in 132 column mode.
`tek`	For plotting on Tektronix 4014 terminals.
`regis`	For plotting on terminals supporting REGIS graphics protocol.
`postscript`	Plots will be formatted for printing on a PostScript-compatible printer. The global variable **`plotoutput`** should be assigned a name of a file before the `plot` command is given. See MRM or **`help(plot,device)`**.
`ln03`	Plots will be formatted for printing on DEC LN03 printer. See MRM or **`help(plot,device)`**.

You may use the familiar **Copy** to place the contents of the active plot window into the Clipboard, or you may save the plot window to a file in PICT format. To

[6] A *cubic spline* is a collection of cubic polynomials chosen to smoothly connect given points of data. They are specially chosen so as to be easy to compute and easy to use in plotting. *Numerical Mathematics and Computing*, 2d ed., by Cheney and Kincaid, (Brooks/Cole, 1985) explains the properties of splines.

do this, activate the window and then use **Save As...** to name and then create the file that will contain the PICT definition of the plot. A plot file can be read into a Maple session by using the **Open** menu item to open the (PICT) file. A plot window will be created to contain the file.

3.7 Other formats for output: fortran, latex, and eqn

Ordinarily, results of Maple are displayed in prettyprinted format. However, other styles of output are possible, as we've seen. By using colons instead of semicolons, you can suppress the printing of results of interactive commands. And as discussed earlier, you can display results in "linear expression syntax" by using `lprint` or into a file via `save`. In this section we discuss alternative formats for output besides those produced by `print` and `lprint`. We also discuss commands other than `save` that cause output to go into a file.

3.7.1 Fortran-style output

The command **fortran**(*expr*); will print an expression using the syntax of the Fortran 77 programming language instead of Maple's. Before using this command for the first time in a Maple session, you must use the command **readlib(fortran);** to load the Fortran translation facility in from the library (see Section 3.12 for a further explanation of `readlib`). If the expression is longer than one line, Fortran continuation conventions are used.

The translation to Fortran does have its idiosyncrasies. For example, the Maple command `fortran(a = sqrt(2)*x);` results in the output `a = sqrt(2)*x`—a Maple equation is translated into a Fortran assignment statement. However, `sqrt(2)` would cause difficulties, since a Fortran `INTEGER` argument is presented to `sqrt`, which accepts only Fortran `REAL` arguments. Another idiosyncrasy is that if the expression being converted is very large, the result may be longer than Fortran's limit of 19 lines.

The command has two options:

fortran(*expr*, **filename**=*name*);

will write the Fortran-style output on the specified file. The option

fortran(*expr*, **optimized**);

takes a single expression and produces a statement sequence in Fortran format that reduces the number of arithmetic operations necessary to compute the expression, through common subexpressions and binary powering. These two options can be combined:

fortran(*expr*, **filename**=`prog.f`, **optimized**);

will write an optimized version of the expression on the file `prog.f`. A readlib-defined function, **optimize**, will display the sequence without translating it to Fortran format first. See **help(fortran);** or MRM for further details and features of `fortran`.

Example 26
Fortran output displayed on the screen

```
>
   expr1 := ln(x)+2*ln(x)^2-ln(x)^3:
>
   readlib(fortran):
>
   fortran(a = expr1);
      a = alog(x)+2*alog(x)**2-alog(x)**3
>
   fortran(expr1,optimized);
      t1 = alog(x)
      t2 = t1**2
      t6 = t1+2*t2-t1*t2
```

3.7.2 Output formatted for LaTeX and eqn

latex and **eqn** are two readlib-defined functions that display expressions in format suitable for entry into text processing programs. Here is an example:

Example 27
Generating a LaTeX version of a matrix

```
>
   readlib(latex):  readlib(eqn):
>
   a := array([[4,(x**2+1)/(sin(x)**2+x**2)],
>
     [3/(sqrt(x)+1/(y+1)), binomial(3,n)]]);
a := array ( 1 .. 2, 1 .. 2,
```

$$\left[\begin{array}{cc} 4, \dfrac{x^2+1}{\sin(x)^2+x^2} \\ 3\left/\left(x^{1/2}+\dfrac{1}{y+1}\right)\right., \mathrm{binomial}(3, n) \end{array}\right]$$

```
)
#latex format output is directed to the file latex.out
#
>
   latex(a,`latex.out`);
#eqn command has no second argument, so eqn format appears in the Session window
#
>
   eqn(a);
left [ matrix { ccol {{4 } above {{ 3  over { { sqrt  "x" } + { {{ {
 "y"  + 1 } )} sup -1 }}}}ccol {{{ { {  "x"  sup 2 } + 1 } over { { {
sin^(  "x" )} sup 2 } + {  "x"  sup 2 }}}} above {{binomial^( {3 ,~
 "n" }}}}} right ]
```

The `latex` command in the example above produces the following result

```
\left[ \begin{array}{ c c } 4&{ \frac{3}{{ ( \sqrt{x}}+ { (( { y+ 1}
 )}^{-1}}}}}\\{ \frac{{ { x^{2}}+ 1}}{{ { ( \sin ( x)}^{2}}+ { x^{2}
}}}}&{3 \choose n} \end{array} \right]
```

in the file `latex.out`.[7] Processing that file with LaTeX results in the output

$$\left[\begin{array}{cc} 4 & \frac{3}{\sqrt{x}+(y+1)^{-1}} \\ \frac{x^2+1}{\sin(x)^2+x^2} & \binom{3}{n} \end{array}\right]$$

The `eqn` command works very similarly to the `latex` command, except that its output is to be used with the *troff* document processor. Neither `latex` nor `eqn` tries to solve the problem of deciding where to break an expression that requires more than one line to print. You must manipulate the results of `latex` and `eqn` by means of a text editor to achieve desired breaks.

3.7.3 Maple session output to a file with `writeto`

`writeto`(*filename*) causes all printed output to be written into the file *filename* instead of the **Maple Session** window until another file-writing command is given. This includes the results of commands and of `print` and `lprint`, error messages, and so on. The command `writeto(terminal)` displays output in the **Maple Session** window again.

3.8 What! You made a mistake? (Syntax errors)

Maple has syntax (grammatical) rules for what it considers valid input. For example, Maple requires you to end calculation entries and commands with semicolons or colons. It won't do anything with your entry until the punctuation mark is entered.

When you enter a calculation or command that does not obey Maple's grammatical rules, a *syntax error message* will be printed out. These messages are of the form **Syntax error before:**, followed by a fragment of the line that you have just entered. The message means that what was ungrammatical about the command occurred somewhere before (usually, but not always, right before) the fragment.

Example 28
A collection of syntax errors

```
#Intended to type "2+2;", but forgot the 2
>
   2+;
Syntax error at the end of the expression
```

[7] Commands that write on files such as `latex` use the same conventions as `save` to change "Maple-style names" into the file names as used by the Macintosh hierarchical file system. See Section 5.11.

```
#Too many (or too few) parentheses
#
>
   (a + b))/2 +c;
Syntax error before: /2 +c;

#Some words are reserved by Maple for its programming language—
#for example, the words "if" and "done".
#
>
   if + done;
Syntax error before: ;

#Sometimes the error occurs far before the context cited.
#Here, the user forgets to enter a semicolon after "2 + 3".
#
>
   2 + 3
>
   2 + 4;
Syntax error before:  + 4;
```

As was mentioned in Section 3.1, you must surround Maple "strange names" with backquotes (`). Since "strange names" can even include the "carriage return" character, Maple will not consider as an error a line that has a starting ` but no ending `. In that situation, it will print out a warning that you are creating a name that is longer than one line.

Example 29
What happens when your backquotes don't come in matching pairs

```
>
   poly := 1 + `A funny name;
Warning: String contains newline character. Close strings with ` quote.
>
   poly;
Warning: String contains newline character. Close strings with ` quote.
>
   What happened?
Warning: String contains newline character. Close strings with ` quote.
>
   oh.  I forgot the other backquote`;
poly := 1

   + A funny name;
poly;
What happened?
oh.  I forgot the other backquote

>
   #The variable "poly" now has a very strange value!
```

● ATTENTION

Remember you can interrupt a Maple computation using the **Halt** menu. See Section 2.7.2 for detailed information.

3.9 You ask too much! (Run-time errors)

Maple is limited to the realm of what is mathematically possible. You cannot get it to divide by zero, or to compute tan $\pi/2$. It is also limited to what its programming and computing resources allow it to do.

When you tell Maple to do something it is not permitted to do, such as to divide by zero, or to create a number or expression that is larger than its limits will permit, it will give you a *run-time error message* beginning with the phrase **Error**,..., or **Error**, in *function*,..., where *function* is the name of the Maple function where the difficulty is occurring. Sometimes you will see that the function named in the error message is not the command you specified, but rather a subsidiary whose computation was supposed to be a step in achieving the overall result.

● ATTENTION

A run-time error message describes only the symptom, not the cause, of an erroneous computation. It is the signal that something was asked for that Maple can't do (e.g., an operation not defined). Such errors are usually corrected by thinking about what you intend to compute and finding the difference between that and what you said.

Example 30
A collection of run-time errors

```
#Examples of run-time errors
#
>   1/0;
Error, division by zero

#Can't differentiate x with respect to "5", since 5 isn't a variable.
#
>   diff(x,5);
Error, wrong number (or type) of parameters in function diff;

#The following causes an error because Maple determines before computing
#that the result could not fit in the computer's memory.  Can you
#estimate how many digits long the answer would be if it were
#displayed?
#
>   12345667890^9876543210;
Error, integer too large in context

#Some run-time error messages do not indicate errors by the user,
#but rather inadequacies in Maple's programming.  For example,
#
>   solve({x^6+3*x+1>2},{x});
Error, (in solve/ineqs) unable to handle these expressions

#indicates that the solver in this version of Maple isn't prepared to
#handle the problem, not that it doesn't have a solution.
```

No matter how much memory your Macintosh has, some computations can require more memory than is available. Poorly posed problems that generate circular definitions are good examples of computations that cause memory-related problems. A simple example is the following recursive definition of a name:

```
x := x + 1;
```

Maple displays the message

```
Warning: Recursive definition of name
```

since x is defined in terms of itself. If you ignore this warning and enter an expression that requires x to be evaluated, the computation ends up chasing its own tail ("infinite recursion"). Each of the never-ending evaluations of x requires some memory in the run-time stack and eventually the stack is filled to its limit. This produces the dialog box of Figure 11 and the computation is terminated forcibly.

When you see this dialog box, try to discover why your last computation overflowed the stack. Clues such as the warning about the recursive definition of a name should help. However, Maple might not always display such a warning, especially if there is a chain of several variables with the second variable defined in terms of the first, the third defined in terms of the second, and so on, and the first variable defined in terms of the last variable.

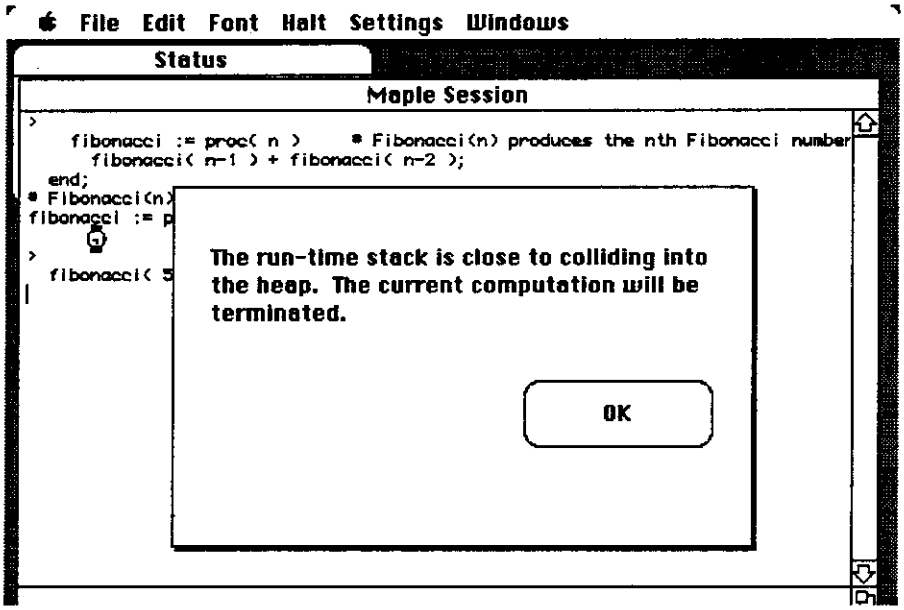

File Edit Font Halt Settings Windows
Status
Maple Session
fibonacci := proc(n) # Fibonacci(n) produces the nth Fibonacci number
fibonacci(n-1) + fibonacci(n-2);
end;
# Fibonacci(n)
fibonacci(5
The run-time stack is close to colliding into the heap. The current computation will be terminated.
OK

FIGURE 11. "Out of stack space" dialog box

Variables defined in a circular fashion are not the only way to exhaust memory. Recursive functions, which are functions defined in terms of themselves, are a powerful and useful concept and often used deliberately. A popular example of a recursively-defined function is that of the Fibonacci sequence: $F_n = F_{n-1} + F_{n-2}$, where $F_0 = 0$ and $F_1 = 1$. If you use the following Maple definition for `fibonacci`:

```
fibonacci := proc(n)
    fibonacci (n-1) + fibonacci (n-2);
    end;
```

you make the common mistake of forgetting to check for the boundary values. Any use of the procedure as it's defined above, for example, the evaluation of `fibonacci(5)`, results in a potentially infinite sequence of recursive function calls and produces the dialog box indicating the run-time stack is filled to its limit.

Complicated but correct programming can also exhaust the memory allocated to the stack. Each Maple function call nested inside another requires some stack memory. If the computation you start is very "deep" there might not be enough room on the stack to finish it. If you suspect this is happening, you can adjust the amount of memory allocated for the stack. Since the stack is set aside when the program is started, you won't be able to change it for your current Maple session. However, you can set the value used when Maple is started again. To do this, select the **Initial stack size...** item from the **Settings** menu. In the displayed dialog box, you will see the current setting for the stack memory. You can edit this value, which is then used for all subsequent Maple sessions.

You might also run out of memory if the computations you start generate many variables and large expressions, though it might not be that the run-time stack is filled to its limit. If this happens, you see the message:

```
System error, ran out of memory
```

You cannot generate new expressions or do further calculations without first cleaning up. Use the garbage collector command, `gc();` to remove any unreferenced expressions (see Section 5.14). It is also advisable to save the results of the session into a file before continuing.

3.10 Saving the state of your Maple session

You can save the state of a Maple session and continue with it later. The command

save `` `*filename*.m` ``;

will record the state of all variables and procedures on your file *filename*`.m`.

After a save command, you can quit the Maple session. The file you've saved to will retain the state of your session. For example, `save` `` `foo.m` ``; will save the state of the session on `foo.m`.

● ATTENTION

Don't forget the backquotes around "strange names" such as `foo.m`. Maple will evaluate an un-backquoted `foo.m` to `foom`.

Later, you can start up another Maple session. Your saved "dot-m" file may now be read back in by two methods with identical results. The first is to choose the **Open** item from the **File** menu to open your file. The second is to use the **read** command in Maple:

read `` `*filename*.m` ``;

will restore all variable and procedure definitions as they were recorded in *filename*.m. You can then proceed as if you had never quit Maple. The command

save *var1*, *var2*, ... *varn*, `` `*filename*.m` ``;

will save only the values of the variables *var1*, *var2*, ... *varn*, instead of everything. In a later session,

read `` `*filename*.m` ``;

will restore the saved values.

Example 31
Retaining Maple results between sessions

Assume your first Maple session consists of:

```
> a:=47;
                                  a := 47

> b:=x^2;
                                       2
                                  b := x
#Save all values in the file FILEALL.m
> save `FILEALL.m`;
#Save the value of a in the file FILESOME.m
> save a,`FILESOME.m`;
```

In a later Maple session, reading in the file `FILEALL.m` recalls the values assigned in the previous session.

```
> read `FILEALL.m`;
> a*b;
                                       2
                                  47 x
```

`FILESOME.m` had only the value of a saved in it. In a subsequent Maple session, we see

```
> read `FILESOME.m`;
> a*b;
                                  47 b
```

3.11 Recording results in files in human-readable format

While internal format is designed to be quickly processed by the computer, it uses a code not easily read by people. Plain text files are useful when you want to print out Maple results for a document.

You can save the contents of a text window (including the **Maple Session** window) to a file by making it the active window and then using the **Save As...** item from the **File** menu. (To activate a window, click anywhere in the window. This should bring the window to the top of your desktop and also highlight the title bar at the top of the window.) **Save As...** will ask you to select a name for the file to be created. If you have already saved a window once, you can simply use the **Save** menu item to update its associated file. The menu items save into a file a verbatim log of the session, as seen in the **Maple Session** window.

You can also use a variant of the Maple `save` command to save your Maple session in plain text: when you give the file name *without* the `.m` suffix, Maple creates a text file in `lprint` format (see Section 3.2). The command

save `` `filename` ``;

saves the entire session into the text file *filename*, and the command

save *var1*, *var2*, ... *varn*, `` `filename` ``;

saves the current values of the specified variables into *filename*.

3.12 Access to additional library procedures

readlib (*procedure_name*);

Many programming systems (not just Maple) rely on a library to provide users with access to commonly used procedures or data. Libraries are usually maintained as a collection of files that any user can access, although they are maintained in a separate location from the files owned by users. Most of Maple's expertise in mathematical calculation is recorded in the Maple *library*, a collection of files separate from the initial contents of your Maple session.[8] When you start a Maple session, the capabilities available consist mainly of the understanding of Maple command syntax and simple arithmetic involving rational expressions. When you specify to Maple some common procedure such as `solve` or `sum`, the part of the library having

[8] Version 4.2's library includes both the executable `.m` files, the programs that generate the library `.m` files, and the files consulted by `help`. There are 1,218 files in all.

to do with that procedure is automatically read in (in the sense of the `read` statement) before the computation commences. For example, when you issue the command

```
solve( x^ - 1 = x , x );
```

the Maple file `solve.m` is read in from the library before the solving computation starts. (`solve.m` in turn automatically reads in other files to handle particular kinds of equations (e.g., systems of linear equations, factoring polynomials, etc.).

You can read in those and other library files directly through **read** commands. For example, the command

```
read ``.libname.`/solve.m`;
```

reads in the library `solve.m` file.[9]

Automatic loading from the library is provided as a convenience for common commands. However, there are portions of the library that you must read in explicitly before using. The **readlib** procedure is a more succinct way of reading library procedures. MRM and the **help** command describe which library procedures must be loaded in via `readlib` before they can be used.

There are various forms of the `readlib` command (see MRM or **help(readlib)** for a complete description). For example, `readlib(fibonacci);` reads the same file as the command `read ``.libname.`fibonacci.m`;`. This is the part of the library associated with the function `fibonacci(`$n$`)`, which computes the n-th Fibonacci number.

Unlike `read`, `readlib` returns a value—the definition of the program that specifies how to compute the function being read from the library. You can suppress such printout by using a colon instead of a semicolon at the end of the `readlib` command.

Example 32
Reading in a library procedure with `readlib`

```
#The library has a procedure for computing Fibonacci numbers.
#It is not read in automatically when you use it, unlike widely used
#procedures such as solve or diff.
#
>
   fibonacci(47);
                                  fibonacci(47)
#The library file can be read in via the readlib command.  We use a colon
#to suppress the printing of the value returned by readlib.
#
>
   readlib(fibonacci):
```

[9] `libname` is a variable with a predefined value, the location of the Maple library for your system. If you use **print(libname);** in a Maple session, you will find out how it is defined for your system. **``.libname.`/solve.m`** is an expression whose value is the complete name of the library file `solve.m`. This complete name is created via the "." operation, concatenation. Concatenation is discussed further in Section 4.8.

```
#And now it can be used.
#
>
   fibonacci(47);
                                2971215073
```

3.13 help yourself to more of Maple

The **help** function allows you to display further details of Maple in a separate **Information for** window. These details include facts about built-in functions such as `factor`, `normal`, and `int`, as well as descriptions of more advanced features.

`help` provides a "thumbnail sketch" of a particular topic. As with a dictionary, you must have some basic idea of where to look for what you are interested in before you can use `help`. This requirement is usually fulfilled by knowing the name of the Maple command or topic you wish to find out about, or by consulting the help index through **help(index)**. The `help` command is designed to be quick to display and read, so it may not be as comprehensive as the explanation found in MRM.

Sometimes a help call involves giving `help` two arguments instead of one. The help index in Example 35 indicates that you should give the command `help(index, library)` to see a list of all the standard library functions.

The command **help()** gives an overview of the available topics, any one of which can be pursued with further `help` commands, as shown in the following example.

Example 33
Overview of on-line help via `help()`

```
#help(); gives an overview of help
>
   help();
```

The resulting **Information for Help** window shows:

```
FUNCTION : help - descriptions of syntax, datatypes, and functions

CALLING SEQUENCE :  help( topic );  or  help( package, topic );

SYNOPSIS :
help( intro );              introduction to Maple
help( index );              list of help categories
help( index, library );     Maple library functions and procedures
help( index, <category> );  list of help files on specific topics
help( <topic> );            explanation of a specific topic
help( <package>, <topic> ); explanation of a topic in <package>
help( `<keyword>` );        explanation of a statement keyword
help( distribution );       for information on how to obtain Maple

Note 1:  A Maple keyword (reserved word) cannot be used as an argument to
the help function unless it is enclosed in backquotes. For example,
"help(quit);" causes a syntax error - use "help(`quit`);" . Note that the
string delimiter is backquote (`) not apostrophe (') nor double quote (").

Note 2:  A command must end with a semicolon, followed by <ENTER>,
before Maple will execute it and display the result. The semicolon can
```

```
appear on the next line if you forget to end the command with it, but it
must appear. There can be multiple commands on one line, separated by
semicolons or colons.

SEE ALSO : keywords, quotes, colons, `quit`
```

Example 34

help for library functions

```
#Help is available on most Maple library commands.
#Help for the "factor" function is shown below.
>
   help(factor);
```

The resulting **Information for factor** window shows:

```
FUNCTION : factor - factor a multivariate polynomial

CALLING SEQUENCE : factor( <expr> );

PARAMETER : <expr> - an algebraic expression

SYNOPSIS :
-    Factor computes the factorization of a multivariate polynomial with
     integer or rational coefficients. Note that any integer content is
     not factored.
-    If <expr> is a list, set, equation, or relation, factor is applied
     recursively to the components of <expr>. If <expr> is a rational
     function, <expr> is first "normalized" - see normal. The numerator
     and denominator of the resulting expression are then factored. This
     provides a "fully-factored form" which can be used to simplify <expr>
     in the same way the normal function is used. However, it is more
     expensive than normal.

EXAMPLES :
       factor( x^2 + 3*x - 4 );  yields   (x+4) * (x-1)
       factor( 1 - sin(x)^2 );   yields   - (sin(x)-1) * (sin(x)+1)
       factor( {x^3+y^3 = 1} );  yields   {(x+y) * (x^2-x*y+y^2) = 1}
```

Example 35

help index of general topics; help on Maple statements

```
#Help is also available on other topics which are not computational commands.
#
>
   help(index);

>
   help(index,library);
```

The resulting **Information for index** window shows:

```
The following categories of topics are available in the help subsystem.

SYNTAX :  help( index, <category> );

SYNOPSIS :  <category> may be any of the following:

-    library:       Index of descriptions for standard library functions
-    packages:      Index of descriptions for library packages of functions
-    libmisc:       Index of descriptions for miscellaneous lib functions

-    statements:    Index of descriptions for Maple statements
-    expressions:   Index of descriptions for Maple expressions
-    datatypes:     Index of descriptions for Maple datatypes
-    tables:        Index of descriptions for tables and arrays in Maple
```

```
-   procedures:    Index of descriptions for Maple procedures
-   misc:          Index of descriptions for miscellaneous facilities
```

And the **Information for library** window shows:

```
HELP FOR : Index of descriptions for standard library functions.

SYNOPSIS :
-   The following are the names of Maple's standard library functions.
    Apply the help function with any of these arguments.
-   Some of these functions are internal (built-in) to the Maple kernel
    while others reside as external (readlib-defined) library functions —
    i.e., functions such as gcd which is initially defined by: gcd :=
    'readlib('gcd')'. However, those that are readlib-defined will be
    automatically loaded when invoked so the Maple user need not know the
    distinction.    The   separate   listings   are   available   via
    help(index,internal) and help(index,external).

    abs          anames       arccos       arccosh      arccot       arccoth
    arccsc       arccsch      arcsec       arcsech      arcsin       arcsinh
    arctan       arctanh      array        assign       assigned     asympt
    bernoulli    binomial     cat          chebyshev    coeff        coeffs
    collect      content      convert      copy         cos          cosh
    cot          coth         csc          csch         define       degree
    denom        diff         divide       dsolve       erf          ERROR
    euler        eval         evala        evalb        evalc        evalf
    evaln        exp          expand       factor       frac         frontend
    fsolve       GAMMA        gc           gcd          gcdex        genpoly
    has          hastype      help         icontent     ifactor      igcd
    igcdex       ilcm         indets       indices      int          interp
    iquo         irem         isolve       isprime      isqrt        ithprime
    lcm          lcoeff       ldegree      length       lexorder     limit
    ln           lprint       map          max          maxnorm      member
    min          modp         mods         msolve       nextprime    nops
    normal       numer        op           order        plot         prem
    prevprime    primpart     print        product      Psi          quo
    radsimp      rand         readlib      readstat     rem          resultant
    RETURN       RootOf       round        rsolve       sec          sech
    sign         signum       simplify     sin          sinh         solve
    sort         sprem        sqrt         subs         subsop       substring
    sum          system       table        tan          tanh         taylor
    tcoeff       testeq       time         trace        traperror    trunc
    type         unames       whattype     with         words        writeto
    Zeta         `$`          `@@`         `@`          `intersect`  `minus`
    `union`

SEE ALSO : readlib, libname, help(index,libmisc); help(index,packages);
```

Because of the way the Maple language is designed, a `help` call for certain reserved words of the Maple language must enclose the word in backquotes—e.g., `help(`+`);`, `help(`done`);`, or `help(`save`);`. However, `help` for most Maple commands (e.g., `solve`, `diff`) does not require backquotes. Chapter 4 introduces more of the reserved words that do require backquotes.

Example 36

Getting help for the reserved word `done`

```
#done is a special word of the Maple language that must be
#backquoted for help to work on it.  Other special words include
```

```
#if, for, proc as explained in Chapter 5.
#
>
   help(done);
syntax error before: );
>
   help(`done`);
```

The resulting **Information for quit** window shows:

```
HELP FOR : The quit statement

SYNTAX : Any one of:  quit,  done,  stop

SYNOPSIS :
-    The  quit  statement terminates the Maple session and returns the user
     to the system level from which Maple was entered.
-    You can also abort the Maple session by choosing the "Terminate Entire
     Session" item under the Halt menu during the middle of a computation.
```

Neither this tutorial nor MRM is revised as often as new versions of Maple appear. However, you can access a description of the changes to Maple since the previous version via the command **help(updates);**.

CHAPTER 4

Advanced Maple

We have seen how to use Maple as a calculator for simple algebraic and numerical computations. In this chapter, we explore how to use Maple in more sophisticated ways.

4.1 A few words to Fortran, Pascal, and BASIC programmers

Unlike computer languages such as Fortran or Pascal, Maple was designed to be used interactively. Simple directions such as "Give the name a to the formula $(x+1)^2$" or "compute $\sum_{i=1}^{n} i^3$" are given through one-line commands. Type declarations, BEGIN/ENDs, and other syntactic paraphernalia are avoided in such "one-step programs." More complicated programs are defined by a Maple function (procedure), as explained in Chapter 5. People who know languages such as Fortran, BASIC, Algol, or Pascal (sometimes referred to as the FBAP family of programming languages) should have little trouble learning how to write simple Maple programs.

A significant difference between Maple and the FBAP family is that Maple is designed specifically for algebraic computation: computation with polynomials, trig functions, equations, and so forth, is supposed to be convenient in it. As a consequence, there are many more symbols with predefined meanings than in the typical "general-purpose" language. For example, Maple places a meaning on `int`, `diff`, `true`, `false`, `solve`, and `taylor` in addition to `if`, `:=`, and `done`. While you can change the meanings of `int` and `diff` (unlike, say, the meaning of `:=`), you must do so carefully, or the built-in mathematical expertise needing integration or differentiation will cease to work for you.

Another way in which Maple differs from conventional programming languages is that a variable x in Maple can mean just that—a symbol with no particular, or perhaps an unknown, value. Combinations of symbols (such as adding them together) form some of the objects that can be printed out or assigned as a value to a programming variable. Unlike many general-purpose languages, which were not designed to support interactive algebraic manipulation, it is perfectly reasonable in Maple to have symbols without assigned values.

Maple is fairly rich in built-in data structures. Besides arrays (which are used to represent vectors and matrices, among other things), it has tables, lists, sets, and sequences, as well as specialized mathematical structures such as sums, products,

and equations. On the other hand, Maple does not have facilities for modularity or further data abstraction beyond what can be done with procedures (subroutines) and self-disciplined programming.[1]

The facilities Maple provides have proven adequate to build a mathematical library of tens of thousands of lines of code, written in Maple's own programming language (explained in Chapter 5). They have sufficed to provide algebraic computing to thousands of users. However, the system continues to change and grow incrementally from version to version.

4.2 Programming variables and mathematical symbols

In Maple, symbols (other than symbolic constants such as `I` and `Pi`) play two roles—one as mathematical variables (unknowns) and the other as programming variables (labels to refer to results already computed). Having assigned and unassigned variables can, and often does, lead to confusion. However, this effect is counterbalanced by the infuriating inconvenience that would result if the two varieties of variables weren't syntactically mix-and-match.

A unifying viewpoint is that *all* variables in Maple are programming variables. The ones used as mathematical symbols have as their value *their own name*; those used as programming variables have something else assigned as their value. Enclosing a variable's name in right quotes—the apostrophe character (`)—gives an expression that is the variable's name, rather than its value, if any. For example, `lprint(`x`)` will print `x`, rather than the value that may be assigned to $x$. If $x$ is being used as a mathematical unknown (no assigned value), then `lprint(`x`)` and `lprint(x)` produce the same result: `x`.

A programming variable can be converted back to a mathematical unknown by assigning it its own name as a value. For example, `x := 5;` makes x a programming variable. `x := `x`;` then converts x back to a symbol. We discuss this idea further in Section 4.5.

4.3 Simplification of expressions

4.3.1 Automatic simplification

Maple is designed to leave most expressions in the form in which you type them, or in which they are created during a computation. There are exceptions to this philosophy where the Maple designers believed that almost all users would want certain transformations automatically performed—for example, turning `0*x` into `0` or `x*2` and `x+x` into `2*x`. Such transformations of expressions are referred to as *simplification.*

[1] Packages, as explained in Sections 4.14 and 5.9, are an example of how a discipline of modularity can be established.

The most noticeable features of Maple's automatic simplification can be summarized as follows.[2]

1. Arithmetic on sums, products, and powers of integers, rational numbers, and floating-point numbers is done automatically.

2. Greatest common divisors are removed from the numerator and denominator of rational numbers.

3. Syntactically identical factors in the numerator and denominator of a rational expression are removed.

4. Products of terms are rearranged so that the constant term is the first term of the product.

5. Like terms in a sum or product are collected—for example, `x*x` → `x^2` or `x+x` → `2*x`.

6. If an expression is a product of a number and a sum of *monomials*,[3] the number is distributed into the sum. For example, `1/2 (x + y)` → `x/2 + y/2`.

7. Maple has built-in knowledge about how to simplify the standard mathematical functions for certain arguments. For example, `cos` $\pi/4 \rightarrow \sqrt{2}/2$, `ln 1` → `0`, `binomial(n,1)` → `n`. Transformations that do not eliminate the function (e.g., $\ln x^2 \rightarrow 2 \ln x$) are typically *not* automatic. Section 4.3.2 discusses how to use Maple's `simplify` command to invoke Maple's built-in knowledge on this second class of transformations.

When Maple determines that a newly entered expression is equivalent to an expression already entered into the system, it will be expressed in the same way as before. For example, `x+y+z`, `y+x+z`, `y+z+x` (as well as the other possible permutations) will all be transformed into one form (probably `x+y+z`, but the form varies from session to session) as part of Maple's automatic simplification. For multivariate expressions, you can use the built-in function **`collect`** to reorder an expression in terms of a main variable or variables. See MRM and **`help(collect)`**.

4.3.2 Simplifying expressions with `simplify`

Maple's automatic simplification does not always put an expression into simplest terms. For example, it does not factor or expand polynomial expressions, nor does it remove greatest common divisors from rational functions (ratios of two polynomials). Many "simplifying" transformations can be expensive to compute; some don't always have the effect of simplifying the answer; and often the notion of "simple" is a matter of taste. Maple leaves these transformations for you to apply as you choose.

`simplify`(*expr*); is the general-purpose simplification command. It will "do the right thing" in many, although not all, situations. For example, it always simplifies rational expressions, turning $x^2 - y^2 - (x + y)(x - y)$ into 0. It will also apply all of the standard kinds of simplifications for the functions given in Table 7. Through `simplify`, you can cause Maple to simplify $\sin^2 x + \cos^2 x$ to 1 or $\log x^2$ to $2 \log x$.

[2] Example 8 in Chapter 2 demonstrates some of these automatic simplification rules at work.

[3] A monomial is a product of symbols or functions. For example, if x and y are mathematical symbols, then xy and $x \sin(x^2 + y)y^2$ are monomials.

TABLE 7
Varieties of transformations available with `simplify`

Variety	Simplification involving	Examples
atsign	functional operators	`a @ <x> → a`
GAMMA	the Γ function	`GAMMA(3/2) → 1/2*Pi^2`
hypergeom	hypergeometric functions	`hypergeom([1],[2],1)` `→ 1/2*exp(1/2)*sinh(1/2)`

power	powers, exponentials, logarithms	`(a^b)^c → a^(b*c)`
		`exp(5*ln(x)) → x^5`
		`ln(x*y) → ln(x) + ln(y)`

Psi	the ψ function	`Psi(1/2) → -gamma-2*ln(2)`
radical	expressions involving radicals (fractional powers)	`(x^2 - 4*x + 2)^(1/4)` `→ (x - 2)^(1/2)`

RootOf	expressions including the `RootOf` function	`RootOf(x^2-2=0,x)^2 → 2`
		`1/RootOf(x^2-2=0,x)`
		`→ 1/2 * RootOf(x^2-2=0,x)`

sqrt	square roots or powers of square roots	`16^(3/2) → 64`
		`(10*x^2 + 60 * x + 90) ^(1/2)`
		`→ 10^(1/2)*(x+3)`

trig	powers of trigonometric functions	`1+tan(x)^2 → 1/cos(x)^2`
		`cos(2*x)+sin(x)^2 → cos(x)^2`
		`cos(x)^2 + sin(x)^2 → 1`

`simplify` always applies the `normal` function as well as these kinds of transformations. The **normal** function is explained in Section 4.4.

`simplify(`*expr*`,` *variety*`)`, where *variety* is something from the first column of Table 7, performs just the specified variety of simplification.

Example 37
Examples of the `simplify` command

```
#Automatic simplification leaves this expression alone.
>
   expr := 2*(cos(2*y))^2 + 3*(sin(2*y))^2 +
>
       exp(x)^(5/2) * (1+2*x+x^2)^(1/2) + 2^(5/2);
                    2            2         5/2             2 1/2    5/2
     expr := 2 cos(2 y)  + 3 sin(2 y)  + exp(x)    (1 + 2 x + x )    + 2
```

```
#But the simplify command employs judicious use of certain
#transformations to simplify the expression.
>
   simplify(expr):
                4          2                                        1/2
       - 4 cos(y)  + 4 cos(y)  + 2 + exp(5/2 x) + exp(5/2 x) x + 4 2

#The second argument to simplify specifies that only one kind of
#simplification should be performed.
>
   simplify(expr,power);
                2          2                          2 1/2    5/2
       2 cos(2 y)  + 3 sin(2 y)  + exp(5/2 x) (1 + 2 x + x )    + 2

>
   simplify(expr,trig);
                  4          2            5/2         5/2      5/2
         - 4 cos(y)  + 4 cos(y)  + 2 + exp(x)    + exp(x)    x + 2

#Below is one of the roots of x^3 -12 * x^2 + 44*x - 48 = 0.  You might
#get this if you were working out the solution by hand using the general
#formula for solution of a cubic equation.
#
>
   -1/2*(-64/27)**(1/6)-1/2 * (-1)**(1/3) * (-64/27)**(1/6)
>
   +  4  +  (-3)**(1/2) * 1/2 * ((-64/27)**(1/6)
>
   -  (-1)**(1/3) * (-64/27)**(1/6));
          64  1/6           1/3     64  1/6
- 1/2 (- ----)     - 1/2 (-1)     (- ----)     + 4
          27                        27

             1/2      64  1/6       1/3     64  1/6
   + 1/2 (-3)     ((- ----)     - (-1)     (- ----)     )
                      27                     27

#Radsimp simplifies the radical subexpressions.
simplify(");
                                    6
```

4.4 Simplification with expand, factor, and normal

expand distributes multiplications over additions. For example,

```
expand((x + y) * (x - y));
```

computes

```
x^2 + y * x - x * y - y^2
```

which is automatically simplified to x^2-y^2. Thus, the result of the expand is $x^2 - y^2$. expand is useful in situations where you think that this kind of term cancellation from turning products into sums leads to a simpler result.

expand also applies multiple-angle rules for trigonometric functions, sum-of-argument rules for the exponential function exp, and expansion rules for other functions.[4]

Example 38
Expansion of multiplication over addition, and of cos, exp

```
>
   expr1 := (x+y+z)^2 - (x+y)^2;
                                        2          2
                     expr1 := (x + y + z)  - (x + y)

>
   expand(expr1);
                                                 2
                              2 x z + 2 y z + z

#expand also knows about multiple-angle rules for trigonometric and
#exponential functions, among others.
#
>
   expr2 := cos(a+b)-exp(a+b):
                     expr2 := cos(a + b) - exp(a + b)

>
   expand(expr2);
              cos(a) cos(b) - sin(a) sin(b) - exp(a) exp(b)
```

There may be situations where the factored result is more succinct than the expanded result, such as $(x+1)^{100}$. Using expand on such an expression would produce a result 101 terms long. Not only is the expanded result difficult to comprehend, but it also takes up much more space in the computer than the factored result. The **normal** function is designed to be an alternative to expansion. It performs simplification while trying to avoid expansion as much as possible.

The general form of a "normalized" expression combines everything into one rational expression, multiplied by a rational constant. The numerator and denominator will be *relatively prime*. The rational constant will be such that the numerator and denominator in the rational expression will have *integer content* 1.[5]

[4] Two library functions, **expandoff** and **expandon**, can be used to suppress the application of such expansion rules if you do not wish to use them. expand also has an optional second argument that controls which expansion rules are applied. See **help(expand)**, **help(expandon)**, or MRM for further details.

[5] Two expressions are relatively prime if they have no common factors (other than 1). The integer content of a polynomial is the greatest common divisor of all the coefficients of a polynomial with integer coefficients.

Example 39
Using normal to simplify rational expressions

```
#Normal can be used to simplify rational expressions, possibly without
#expanding numerators and denominators.
>
   (2*x+4*y)*(x^2-y^2)/(x+y)^2;
```

$$\frac{(2\,x + 4\,y)\,(x^2 - y^2)}{(x + y)^2}$$

```
>
   normal(");
```

$$2\,\frac{(x + 2\,y)\,(x - y)}{x + y}$$

```
>
   normal(x^2/(1-x)^2 - x/(1-x)^2);
```

$$\frac{x}{-1 + x}$$

```
#Normal form ensures that a rational expression equivalent to
#zero is normalized to zero.
#
>
   normal( (x^2-y^2) - (x+y)*(x-y));
```

$$0$$

```
#Normal form is not always "most succinct" form, however.
>
   2*x/(x+y)^3 + x/(x+y)^2 + y/(x+y) + y/(x+y)^0;
```

$$2\,\frac{x}{(x + y)^3} + \frac{x}{(x + y)^2} + \frac{y}{x + y} + y$$

```
>
   normal(");
```

$$\frac{2\,x^2 + x^2 + x\,y + x^2\,y + 2\,x\,y^2 + y^3 + x^3\,y + 3\,x^2\,y^2 + 3\,x\,y^3 + y^4}{(x + y)^3}$$

factor(*expr*) will perform a complete factorization of polynomials in one or more variables (with respect to the coefficient field of rational numbers). While modern algorithms have led to a reasonably efficient factorization algorithm, the process of factorization is a relatively expensive operation compared with expand or normal, so it should be used conservatively.

Example 40
Factorization of polynomials over the integers via `factor`

```
#Example of univariate factorization
>
   expr := x^3 + 3*x^2 + 3*x + 1;
                                    3      2
                          expr := x  + 3 x  + 3 x + 1

>
   diff(expr,x);
                                      2
                                 3 x  + 6 x + 3

>
   factor(");
                                        2
                                   3 (x + 1)

#Compare factored form of derivative to derivative of factored original.
#They should be the same!
>
   factor(expr);
                                       3
                                  (x + 1)

>
   diff(",x);
                                        2
                                   3 (x + 1)

#Example of multivariate factorization
#
>
   x^4*sin(y)^4 + x^4*sin(y)^3 + x^2 * sin(y) + x^2;
                     4       4   4       3   2           2
                    x  sin(y)  + x  sin(y)  + x  sin(y) + x

>
   factor(");
                          2                2      3
                         x  (sin(y) + 1) (x  sin(y)  + 1)
```

4.5 Full and delayed evaluation

Consider the sequence of assignments in the following example.

Example 41
"Full evaluation" follows the chain of assignments

```
>
   x := y;
                                    x := y

>
   y := z;
                                    y := z

>
   z := 5;
                                    z := 5
```

```
>
   x;
                                5
```

When x was first used as a programming variable, it had the value y. Yet 5 is the value printed out for x on the last line of the example. Whenever Maple does a calculation, the symbols in the expression are checked to see whether they are being used as programming variables (that is, whether any of them have been assigned values). If that is the case, Maple substitutes the assigned value for the symbol in the expression. It then checks to see whether there are more value/programming variable substitutions possible and does them as well. The result, after there are no more substitutions possible, is called *full evaluation.*

A problem many users of Maple encounter is due to full evaluation. It occurs whenever they use i as a programming variable and then as the index of summation in a sum computation.

Example 42
A problem due to full evaluation

```
#Use i as a label for a constant in a calculation.
>
   x := 4:       i := 5:
>
   x^i!;
   1766847064778384329583297500742918515827483896875618958121606201292619776
#Now think of i as a mathematical symbol.
>
   sum(i^2,i=1..n);
Error, (in sum) invalid arguments
#
#i has a value left over from before!  It can't be a mathematical symbol.
#
>
   i;
                                5
```

What was the error? i was first used as a programming variable—that is, as a label for a computed result. When the sum was computed, Maple first applied full evaluation to the arguments of the sum procedure. Thus, sum(i^2,i=1..n) turned into sum(25, 5=1..n) before the sum procedure was invoked. The error message was generated because "5" is not a proper summation index. We explain how to resolve this common situation in Example 45 in Section 4.6.

While it might be argued that sum should be designed not to use "full evaluation," the designers of Maple have opted for consistency, since it is easier to remember a general rule across many functions than to keep track of an assortment of cases. Thus, the policy of "full evaluation" has only four exceptions:

1. The **assigned** and **evaln** functions evaluate their arguments only to the point where they become names (i.e., single variables) and no further. See MRM or **help** for further details about these functions.

2. Maple procedures, tables, and arrays have a special kind of evaluation called last-name evaluation. The result of evaluation is not the procedure or array itself, but the programming variable that is assigned the procedure, table, or array as its value. This will be illustrated further in Sections 4.11.1 ("Tables and arrays") and 5.5 ("Simple Maple procedures").

3. The concatenation operator (.), discussed in Section 4.8, does not evaluate the first item in a concatenation expression. But it is a peculiarity in other ways, since concatenation expressions are one of the few types of expression permitted on the left-hand side of an assignment operator—for example, `a.i := i;`.

4. The local variables of a procedure use *one-level evaluation*. If a local variable has been assigned, then the result of evaluation is the most recently assigned value; otherwise, it is the unassigned name. This point is discussed in Section 5.6 ("More Maple procedures").

4.6 Quotation and unevaluation

Enclosing a variable in right quotes (´) is the way to specify the *name* of a programming variable instead of its *value*. Quoting a variable or an expression prevents evaluation of the variables enclosed by the quotes. However, quoting does not prevent simple arithmetic or most of the rest of the operations of the automatic simplifier (see Section 4.3.1)—for example, $1 - 1 \rightarrow 0$, $y/y \rightarrow 1$. The result of "fully evaluating" a quoted expression is to strip off one layer of quote marks.[6]

Example 43
The effect of evaluation, levels of quotation

```
#x used as a programming variable.
>   x := 3;
                                    x := 3

#Quoting x gives a polynomial in "x" instead of 4.
>   'x'+1;
                                    x + 1

#Full evaluation of a quoted expression or variable
#strips off one level of quotation.
>   '''x'+1'';
                                  '('x') + 1'

>   '';
                                   ('x') + 1

>   '';
                                     x + 1
```

[6] If you want to find out how to prevent even simple arithmetic simplifications, see "Unevaluated Expression" in MRM.

```
>
   ";
                                    4

#Quoting an expression delays substitution of the values of the
#programming variables and functions within it, but does not prevent
#arithmetic.
>
   '(3 + x - 3 + 3*y/y) mod x';
                                (3 + x) mod x
```

A special case of "unevaluation" arises when you wish to remove the assigned value from a programming variable—that is, to restore it to being a mathematical symbol. To do this, assign the variable its name—for instance, `x := 'x';`.

Example 44
Assigning and unassigning variables

```
#Assign x the value 3, and print out the value of x+1.
>
   x := 3: x + 1;
                                    4

#Unassign x — restore it to being a mathematical symbol.
>
   x := 'x';
                                  x := x

#Print out the value of x+1.
>
   x+1;
                                  x + 1
```

● ATTENTION

To resolve the situation with `sum` and `i` mentioned in Example 42 (Section 4.5), you could remove any assigned value from `i` or quote `i` wherever it appears in the `sum` command. Sometimes it is a useful precautionary measure to quote, or to unassign, a variable in an expression or command if you wish to use it as a mathematical unknown with no value but aren't sure whether it has previously been assigned a value in the Maple session.

Example 45
Resolving the problem with `i` and `sum`

```
#Use i as a label for a constant in a calculation.
>
   i := 5:
#"5" can't be used as a variable of summation because it has a value.
>
   sum(i^2,i=1..n);
Error, (in sum) invalid arguments
```

```
#i has a value left over from before!  It can't be a mathematical symbol.
#
>
   i;
                                        5

#Remove the assignment of i.
>
   i := 'i';
                                    i := i

>
   sum(i^2,i=1..n);
                          3              2
                1/3 (n + 1)  - 1/2 (n + 1)  + 1/6 n + 1/6
```

4.7 Using quoted variables as function arguments

Maple functions follow the mathematical tradition that functions are single-valued. Sometimes Maple functions bend this notion a bit and return a set of expressions, but since a set is a valid Maple entity, this result is still one "value."

Yet sometimes it is desirable to return several results through separate channels. Quoted variables are often used as additional arguments to Maple functions in order to export extra information from them. Certain Maple functions follow this programming rule—optional extra arguments to a function are for exporting extra information. These optional arguments must be quoted or unassigned variables. During the course of computing the function's value, these export variables are assigned additional information that the function computes and exports.

divide is a built-in Maple function that exports extra information through additional arguments. `divide(`*p*`,`*r*`)` computes the quotient of two polynomials, *p* and *r*, that are given as the arguments to the procedure. The value returned by `divide` is `true` if *r* divides *p*, `false` otherwise. During the course of determining whether *p* divides *r*, the quotient *p*/*r* is formed, but there is no way to return it as the value of the function, since either `true` or `false` is being returned through that channel. However, you can give an optional third argument to `divide`, which must be either a quoted variable or a variable with no assigned value. The third argument is assigned the quotient if the value of `divide` is `true`. In this way, "full evaluation" can be applied uniformly to all arguments of the function, yet auxiliary information can still be passed back through some of them.

Example 46
Use of quoted variables as parameters to library procedures

```
#Use a as a programming variable.
>
   a := 3:
>
   p := x^2 - 1:
>
   q := x+1:
```

```
#divide returns true if the second argument divides the first over the
#rationals.  It assigns the third argument the value of the quotient.
#The third argument should be either an unassigned variable or a quoted one.
>
   divide(p,q,b);
                                        true

>
   b;
                                      x - 1

#In the divide computation below, full evaluation of the arguments means
#divide(x^2-1, x+1, 3); is being specified.  Assignment of the quotient
#to "3" is impossible, since "3" isn't a variable.
>
   divide(p,q,a);
Error, wrong number (or type) of parameters in function divide;

#'a' yields the name of the programming variable a.  Thus the following
#example calls divide with the arguments  (x^2-1, x+1, a).
#
>
   divide(p,q,'a');
                                        true

>
   a;
                                      x - 1
```

4.8 Concatenation: Forming new names from old

In Maple, concatenation is the joining together of two or more symbols to produce a single symbol.[7] The way concatenation is specified is through the "." operator. For example, if `u`, `t`, and `k` are symbols with no assigned values, then the expression `u.t` evaluates to the symbol `ut`, and the expression `u.t.k` evaluates to the symbol `utk`. If the result of concatenation is a name, then you can have a concatenation expression on the *left-hand side* of an assignment.

Concatenation follows the full evaluation rule *except* that the leftmost symbol is not evaluated. The following example illustrates what happens with the concatenation of symbols with values.

Example 47
Concatenation of symbols into a name

```
#Concatenation of mathematical symbols creates one name by joining
#together several.
>
   a . b;
                                        ab

>
   a . b . c;
                                       abc
```

[7] We first saw the concatenation operator in Section 3.10. We backquoted expressions there to *avoid* concatenation.

```
#If the symbols have assigned values, then full evaluation applies
#for all but the first symbol.
>
   a:=x;
                                    a := x

>
   b:=2;
                                    b := 2

>
   c:=y;
                                    c := y

>
   concatresult := a . b . c;
                              concatresult := a2y

#If some symbols do not evaluate to symbols, then an expression
#called an "unevaluated concatenation" is formed.
>
   c := y + 1;
                                  c := y + 1

>
   concatresult := a . b . c;
                             concatresult := a2.(y + 1)

>
   y:=5;
                                    y := 5

>
   concatresult;
                                     a26

#If the concatenated result is a variable name, then one can use it in
#an assignment statement on the left-hand side.
#
>
   a.b := a * b;
                                  a2 := 2 x

#`` (the name with no characters in it) can be used when you need
#evaluation of all symbols in a concatenated expression.
>
   `` . a . b . c;
                                     x26

#The built-in library function "cat" also concatenates using
#full evaluation.
>
   cat(b,a,c);
                                     2x6

#A succinct way to create the expression a0 + a1*x + a2*x^2 + ... +
#a8*x^8.   Quotation is required in this example to prevent the
#premature evaluation of a.i before i takes on specific values.
#
>
   sum( 'a.i*x^i', i = 0..8);
                          3       3       4       5       6       7       8
        a0 + a1 x + 2 x  + a3 x  + a4 x  + a5 x  + a6 x  + a7 x  + a8 x

# 1.3 and 1 . 3 both refer to the floating-point number.  In this
#case, the dot is not a concatenation operator but rather
#a decimal point.
#
>
   1.3 * 1 . 3;
                                    1.69
```

```
#Using the if statement (discussed fully in Chapter 5), we can
#demonstrate the difference between the name `13` and the number 13.
>
  a:= ``.1.3;
                                  a := 13

>
  if a=13 then print(yes) else print(no) fi;
                                     no

>
  if a=`13` then print(yes) else print(no) fi;
                                    yes
```

4.9 Looking at parts of expressions: op, nops, coeff

Every Maple expression can be categorized as being of one or more *types*. For example, x+y+z is a sum (referred to as `+` to avoid confusion with the sum command), while 2*x*y is of `*` type. There are distinct types for the other conventional mathematical objects Maple can represent, as described in Table 8.

Every Maple expression is kept in a data structure. If the data structure contains several pieces of information, the structure may have parts. At the top level of information, these parts are arranged in a linear order; that is, there is a first part, a second part, and so on. However, each part may be a complete data structure itself.

For example, x+y+z is represented in a data structure. The data structure can be thought of as having three parts in which to hold the information about the terms of the sum—in this case, the names (symbols) x, y, and z. 2*(x+y)*z also has three parts, but its second part consists of a sum data structure that itself has two parts.

Maple has several functions to allow you to get at the pieces and type of any expression.

op(*i*, *expr*) refers to the *i*-th part of an expression. For example, op(1, 2*x*y) is 2.

nops(*expr*) is the number of parts (operands) in the expression. For example, nops(2*x*y) is 3, while nops([2,3]) is 2.

whattype(*expr*) returns the description of what type of expression Maple thinks you have. whattype(2*x*y) is `*`.

type(*expr*, *typename*) returns true or false, depending on whether the expression *expr* is of type *typename*. In addition to the whattype typenames, there are types that combine the various categories. For example, type(2*x*y, `*`); is true. type(2*x*y, polynom) is also true, since 2*x*y is a polynomial as well as a product. In Section 4.10.2, we will see how other Maple data structures such as lists and sets have types and op parts. See MRM or **help(type)** for a complete listing and definition of Maple types.

TABLE 8
Types of mathematical objects in Maple

Object type; Instances	Maple type name, whattype result	Examples
number;		
integer	`integer`	`5`
rational	`fraction`	`1/3`
floating-point	`float`	`4.7`
algebraic expression;		
variable	`string`	`x`
sum	`'+'`	`x+y`
		`x-y`
product	`'*'`	`x/y`
		`x*y`
power	`'^'`	`x^5`
function	`function`	`sin(1)`
		`f(x,y)`
		`w!`
composition of functions	`function`	`sin@<x^2|x>`
		`f@@5`
user-defined infix operators	`function`	`a &+ b`
Boolean;		
and	`'and'`	`a and b`
or	`'or'`	`a or b`
not	`'not'`	`not b`
inequalities;		
less than	`'<'`	`a<b; b>a`
less than or equal	`'<='`	`a<=b; b>=a`
not equal	`'<>'`	`a<>b`
equation	`'='`	`a=b-3`
range	`'..'`	`1..10`
series	`series`	result of `taylor`
procedure	`procedure`	`<x^2 | x>`
		`proc(x) x^2 end`
matrix (array)	See Section 4.14, "Linear algebra in Maple."	

See MRM, `help('@')`, and `help(operators)` for a further explanation of `@` and `@@`, user-defined infix operators, and the `<...|...>` notation for defining functions.

● ATTENTION

The names of types can be assigned too, as they are no different from any other symbol names. Thus, if you use

```
integer := 5;
```

to label an integer result for further use, and subsequently enter

```
type(36, integer);
```

"full evaluation" turns this into

```
type(36, 5);
```

This causes an error, since "5" is not a predefined type. Assignment to "system-reserved" names such as `integer` or `polynom` should be avoided, since even if you avoid using `type`, a built-in command might use it in its internal programming. Maple has over 300 such names (see Appendix). If you are unsure about whether you have used type names as variables, you can quote them to be safe, as in

```
type(36, 'integer');
```

Example 48

Types and parts of expressions via `whattype` and `op`

```
> s := x+2*y+sin(3*z^2);
                                                  2
                          s := x + 2 y + sin(3 z )

#whattype describes the type of its argument.
> whattype(s);
                                    +

> whattype(t);
                                  string

#In its simplest form, the type function has value true if its
#first argument is of the type described by its second argument.
> type(1/2, 'integer');
                                  false

> type(5, 'constant');
                                   true

> type(x, 'name');
                                   true
```

```
#nops(expr) is the number of operands in the expression.
#There are three terms in the sum s.  Two of the terms have subterms.
>
   nops(s);
                                    3

#op(i, expr) is the i-th operand of the expression.
>
   op(1, s);
                                    x

#operand 0 for a function invocation is the name of a function.
>
   nops(op(3,s));
                                    1

>
   op(0, op(3, s));
                                  sin

>
   op(1, op(3, s));
                                     2
                                  3 z

#Asking for an operand that isn't there will result in an error.
>
   op(2, op(3, s));
Error, improper op or subscript selector

#Quotients are represented as products.  Products always have the rational
#constant as the first operand (except when the constant is 1).
>
   q := x/(6*y);
                                q := 1/6 x/y

>
   type(q,`*`);
                                  true

>
   op(1,q);
                                  1/6

>
   op(2,q);
                                   x

>
   r := op(3,q);
                                r := 1/y

>
   type(r,`^`);
                                  true

>
   op(1,r);
                                   y

>
   op(2,r);
                                  -1

>
   `product without rational constant` := x*y;
                product without rational constant := x y

>
   op(1,`product without rational constant`);
                                   x
```

```
>
   op(2,`product without rational constant`);
                                    y
```

For a mathematical expression in expanded form (e.g., one that has been processed by the expand command), **coeff**(*expr*, *x*, *n*) is the coefficient of x^n in *expr*. **lcoeff**(*expr*,*x*) is the leading coefficient of the polynomial in x. lcoeff(*expr*, [*x*, *y*]) returns the leading coefficient of the polynomial in x and y, where x is considered the main variable. An error is given if *expr* is not a polynomial in the specified variables. **tcoeff** works similarly to lcoeff for finding the trailing coefficients of polynomials.

Example 49
Finding coefficients of variables with coeff, lcoeff, and tcoeff

```
>
   p := 3*x^4*y^4 + 2*x^4*y^3 + 2*x^3*y^3*z - y^2*z^5 + x*y^5;
                      4  4     4  3     3  3     2  5     5
             p := 3 x  y  + 2 x  y  + 2 x  y  z - y  z  + x y
>
   coeff(p, x, 4);
                                  4     3
                               3 y  + 2 y
>
   coeff(p, cos(y), 1);
                                    0
>
   coeff(p, cos(y), 0);
                    4  4     4  3     3  3     2  5     5
                 3 x  y  + 2 x  y  + 2 x  y  z - y  z  + x y
>
   coeff(p, z, 0);
                             4  4     4  3     5
                          3 x  y  + 2 x  y  + x y
# The leading coefficient of p considered as a polynomial in x.
>
   lcoeff(p, x);
                                  4     3
                               3 y  + 2 y
# The leading coefficient of that result considered as a polynomial in y.
>
   lcoeff(p, [x, y]);
                                    3
# The other order does not produce the same result.
>
   lcoeff(p, [y, x]);
                                    1
```

4.10 Expression sequences, sets, and lists

This section discusses three of the fundamental Maple data structures: expression sequences, sets, and lists. However, there are many other types of data structures as well, besides the mathematical ones described in Table 8. They are listed in Table 9.

TABLE 9 General-purpose data structures in Maple

Object type	Maple type name	Examples
array	**array** or **table**	`a := array(1..10);`
table	**table**	`t := table();`
concatenation	**'.'**	`a.(m+1)`
expression sequence	**exprseq**	`1, b, c`
name	**name**	`x`
list	**list**	`[a,b,c]`
set	**set**	`{a,b,c}`
procedure definition	**procedure**	`proc(x,y) (x+y)^2 end;`
		`< (x+y)^2 | x,y>`
unevaluated expression	**uneval**	`''x''`

4.10.1 Expression sequences

An *expression sequence* is simply an ordering of Maple objects separated by commas, such as `a, b, c`. Sequences can be assigned to variables as values, just as with any other Maple result.

Sequences arise often in Maple, in part because sets and lists (see Section 4.10.2) can be built out of sequences. Some Maple built-in functions return sequences for their results. For example, the `solve` function will sometimes return a solution sequence when there is more than one solution. When the `op` function is given one argument, it returns a sequence of *all* the operands of its argument. The arguments to a function are also a sequence.

The special symbol **NULL** stands for the empty sequence, the sequence with nothing in it. Full evaluation applies to the construction of sequences. Thus, if `seq1` is assigned the sequence of unassigned variables `a,b,c`, then `seq1, d` is the sequence `a,b,c,d`, while `seq1, NULL, seq1, NULL` is the sequence `a,b,c,a,b,c`.

Since sequences occur so often in practice, there is a built-in **$** operator that efficiently creates a sequence parameterized by an index variable varying through a range of integers.

When *lo* and *hi* are integers,

expr `$ i =` *lo* `..` *hi*

constructs the sequence

subs(i =*lo*,*expr*), subs(i=*lo*+1,*expr*), ... subs(i=*hi*, *expr*),

similarly to the way in which sum(*expr*, i=*lo*..*hi*) constructs a definite sum.

Example 50

Sequences and the $ operator

```
# Example of how to build up a finite sequence bit by bit
>
   seq1 := sin(x), x=0;
                              seq1 := sin(x), x = 0
>
   seq1 := seq1, 5;
                             seq1 := sin(x), x = 0, 5
# Sequences can be used as an argument to a function.
>
   t := taylor(seq1);
                                        3      5
                              t := x - 1/6 x  + O(x )
# The $ works much like sum.
#
# Create a sequence of the numbers 12! through 0!
>
    (12-i)!$ i = 0..12 ;
 479001600, 39916800, 3628800, 362880, 40320, 5040, 720, 120, 24, 6, 2, 1, 1
# The series data structure has powers and coefficients as its
# operands.  Quotes are used here to prevent the premature attempt
# to evaluate the op function before i takes on specific values
# (see Section 4.6, ``Quotation and unevaluation'').
#
>
   'op(i,t)'$ i = 0..nops(t) ;
                            x, 1, 1, -1/6, 3, 0(1), 5
# Create a sequence of the 0th, 2nd, and 4th operands of the series
# above, using the built-in seq function.
#
>
   'op(2*i,t)'$ i = 0..2;
                                    x, 1, 3
# op(expr) -- op with only one argument -- returns a sequence of the
# operands of the expression, equivalent to  op( 1..nops(expr), expr ).
#
>
   op( x + y + z );
                                    x, y, z
# op(i..j,expr) is a sequence of the ith through jth operands of expr.
>
   op(1..4, t);
                                 1, 1, -1/6, 3
```

4.10.2 Sets and lists

Sets, lists, tables, and arrays are other ways of collecting information in Maple. We discuss sets and lists in this section; tables and arrays are discussed in the next section.

A Maple *set* is written as a sequence of objects surrounded by braces, { }, the usual mathematical notation for a finite set. Like most Maple objects, sets can be assigned as values to variables and can appear as an argument to, or result of, a function. Maple has built-in operators for set unions, intersections, and set differences, as shown in Table 10.

TABLE 10
Set operations in Maple

Mathematical operation	Maple operation
$a \cup b$	`a union b`
$a \cap b$	`a intersect b`
$a - b$ (set difference)	`a minus b`
is $x \in a$?	`member(x,a)`

In Maple version 4.2, the user cannot specify the printed order of objects in the set. The ordering of elements of sets is determined "by the system." As with the ordering of terms in a sum, the ordering of a set is consistent within a session, but varies from session to session. Thus, in one session, a set may be printed {x,y,z}, while in another it may appear as {z,y,x}. If you wish to preserve the order, use lists instead.

Example 51
Set operations

```
> set1 := {a, b};
                          set1 := {a, b}

> set2 := {a,d,e};
                        set2 := {a, d, e}

> set3 := {b, e};
                          set3 := {b, e}

> u := set1 union set2;
                        u := {a, b, d, e}

> i := set2 intersect set3;
                             i := {e}

> d := set1 minus set3;
                             d := {a}
```

Like expression sequences, a Maple *list* preserves the order of data in the way the user originally specifies. Duplicate copies of the same value are preserved in lists, whereas they are eliminated in sets. On the other hand, the operations of union, intersection, and difference are not defined for lists in Maple.

A Maple list can be created by surrounding a sequence of Maple objects by left and right brackets, [and]. To illustrate the distinction between sequences and lists in Maple, `[a,b], [b,c]` remains as a sequence of two lists, while `a,b , b,c` would automatically be merged into a single sequence `a,b,b,c`. One use of lists is to group objects together when they are to be considered as a single argument to a function.

The `op` function works for sets and lists as one might expect: the i-th operand of a set or list will be the i-th element of the object, with the ordering based on how you would see the set or list printed out. A Maple idiom for operations involving lists (or sets) is to create a sequence that you then turn into a list by wrapping brackets (or braces) around it. (See Example 52.)

An alternative notation for referring to a particular operand of a list, set, or sequence `L` is to use `L[`*integer*`]`, which is referred to as a *selection operation*. `L[`*range*`]` is an alternative notation for specifying a sequence of operands.

Example 52

Merging lists, reversing a list

```
>
   list1 := [1, 2, 3];
                              list1 := [1, 2, 3]
>
   list2 := [4, 5, 6];
                              list2 := [4, 5, 6]
#Concatenating two lists together — op(list) is the sequence of the
#elements of the list.
#
>
   list3 := [op(list1), op(list2)];
                           list3 := [1, 2, 3, 4, 5, 6]
#Reversing a list.
>
   n := nops(list3);
                                     n := 6
>
   list4 := [ 'op(n-i, list3)'$ i = 0..(n-1)];
                           list4 := [6, 5, 4, 3, 2, 1]
#If L is a list, L[i] is another way to refer to op(i,L).
#Here is alternative notation to reverse a list.
#
>
   list5 := [ list4[n-i+1]$i=1..n];
                           list5 := [1, 2, 3, 4, 5, 6]
#The "sequence idiom" also works for sets.
>
   set6 := {list5[1..4]} intersect {op(list1)};
                                set6 := {1, 2, 3}
```

4.11 Tables and arrays: Indexed collections of data

4.11.1 Tables

Maple has *tables* for collecting information in a tabular form. In Example 53, we create a Maple table like that shown in Table 11. The table entries are numbers; each table entry is indexed by a color. Conceptually, we are thinking of the table as an association between colors and numbers representing the wavelengths (in nanometers) of the colors.

TABLE 11
Wavelength table

Color	Wavelength (nanometers)
red	610
blue	480
violet	450
green	520

A particular table entry has an *index* that describes its position within the table. A particular value within the table is accessed by means of a selection operation using the notation *tablename*[*index*], where *tablename* is the name of the table and *index* is a name or an expression sequence. The index is sometimes referred to as the *subscript* in the selection operation.

In Maple, the table creation function is **table**. A list of equations relating the indices and the table values can be an argument to the `table` function. If no arguments are given to `table`, the table will initially be empty. Any Maple programming variable can be assigned a table as its value. Once assigned, that variable becomes the name of the table. The contents of an entire table can be displayed via `print`.

Example 53
Creating and modifying tables

```
#Assign the variable Wavelength a table object with some predefined
#entries.  The indices of the table are colors; the components, numbers
#(wavelength in nanometers).  We assume that the names of the colors
#have no assigned values, or else we would have to quote them.
#
>  Wavelength := table([red=610, blue=480, violet=450, green=520, black=0]);
Wavelength := table (
   [(violet) = 450,
   (black) = 0,
   (green) = 520,
```

```
   (red) = 610,
   (blue) = 480])
#Access a particular table entry.
>
   Wavelength[red];
                                    610
#Add another entry to the table by assigning the appropriate indexed
#component.
#
>
   Wavelength[orange] := 600;
                          Wavelength[orange] := 600
#Alter the table by replacing a component.
>
   Wavelength[red] := 640;
                           Wavelength[red] := 640
#Delete an element from the table by unassigning as one would an
#unsubscripted programming variable.
#
>
   Wavelength[black] := 'Wavelength[black]';
                  Wavelength[black] := Wavelength[black]
#To display an entire table, use print.
>
   Wavelength;
                                 Wavelength
>
   print(Wavelength);
table (
   [(violet) = 450,
   (green) = 520,
   (red) = 640,
   (orange) = 600,
   (blue) = 480])
```

Tables can have heterogeneous indices—numbers, variables, expressions, sequences, or any other valid Maple object. For example, assigning `Memo[Monday]` a value will create a table with the entry for `Monday` in it and assign it to the variable `Memo`, if `Memo` doesn't already have a table for its value. If `Memo` does have a table for its value, then that table is updated.

Example 54
Another example of a Maple table

```
>
   Memo [Monday] := `Nancy's birthday`:
>
   Memo [1, 2 ] := `buckle my shoe`:
>
   Memo [e=mc^2] := `Einstein's formula`:
>
   Memo [g] := [`gravitational acceleration`, 9.8, 32]:
```

```
>
   print(Memo);
table (
   [(Monday) = Nancy's birthday,
   (1, 2) = buckle my shoe,
          2
   (e = mc ) = Einstein's formula,
   (g) = [gravitational acceleration, 9.8, 32]])
```

If the variable a is assigned a table as value, then the action b := op(a) assigns b the *same table*. Later alteration of the table through either a or b alters the single version of the table. (In computer science terms, a and b point to the same object.) If a is assigned another table (or other Maple object, such as a sum or an equation), then b will still have a table as a value. This would not have been the case if we had just done b := a;, which makes b take on whatever the value of a is.

● ATTENTION

Evaluation of programming variables assigned tables as values works differently than the usual "full evaluation." This exception to "full evaluation" is called last-name evaluation. With last-name evaluation, the result of evaluation is the symbol that was the last in the chain of values just before the table was reached. Thus, if you assign the programming variable `Wavelength` a table as its value and then perform the assignment `W12 := Wavelength`, the result of evaluating `W12` is (the symbol) `Wavelength`. However, the table will be the result of evaluating `op(W12)`, and `print(W12)` will display the table (see Example 55).

Example 55
Tables as assigned values, last-name evaluation

```
#Create a table as before.
>
   Wavelength := table(['red'=610, 'blue'=480, 'violet'=450, 'green'=520,
>          'black'=0]);
Wavelength := table (
   [(blue) = 480,
   (violet) = 450,
   (black) = 0,
   (green) = 520,
   (red) = 610])
```

```
#W12 and W13 have the same table as their value, but only the last name
#in the chain of assignments is displayed, not the table itself.
#
>
   W12 := Wavelength:     W13:=W12:
>
   W12;
                                  Wavelength

>
   W13;
                                  Wavelength

#The value of op(tablename) is the table.
>
   op(W13);
table (

   [(blue) = 480,

   (violet) = 450,

   (black) = 0,

   (green) = 520,

   (red) = 610])

#print(tablename) will print the table, although the result of the
#print function is NULL, not the table.
>
   print(W13);
table (

   [(blue) = 480,

   (violet) = 450,

   (black) = 0,

   (green) = 520,

   (red) = 610])
```

To create a distinct version of the table that will not be affected when the original is altered, use the built-in **copy** function.

Example 56
Copies of tables

```
#Create a table of the first few Fibonacci numbers.
>
   fibonacci_table[1] := 1:
>
   fibonacci_table[2] := 1:
>
   fibonacci_table[3] := 2:
>
   fibonacci_table[4] := 3:
>
   fibonacci_table[5] := 5:
>
   fibonacci_table[6] := 9:
```

```
#Assign b the same table, assign c a (separate) copy of it.  "copy" is a
#built-in function for creating a separate copy of a table or array.
>
   b := op(fibonacci_table):
>
   c := copy(fibonacci_table):
#Alter a component of the table assigned to both "b" and "fibonacci_table".
>
   b[6] := 8;
                                    b[6] := 8
#The table, as referenced by the other name, is changed.
>
   fibonacci_table[6];
                                       8
#But the separate copy is unaltered.
>
   c[6];
                                       9
```

4.11.2 Arrays

Arrays are a restriction of tables. An array can have only *integer* subscripts. The range of permissible subscripts must be stated when the array is created via the **array** function, which is similar to `table`. In creating an array, you must specify the index range of integers for each dimension of the array. In addition, you can specify whether the indexing of the array is sparse, symmetric, antisymmetric, or some other possibility of your own design.

Changing and copying arrays works as with tables. What distinguishes arrays from tables in general, other than restriction to integer indices, is that each reference to an array element is checked to see that it is within the index range specified when the array was created.

Example 57
Maple arrays

```
# Define a to be an array with 2 rows and 2 columns.
#
>
   a := array(1..2, 1..2);
                        a := array ( 1 .. 2, 1 .. 2, [] )
# Entries of a can be assigned values.  Unassigned elements of an array
# retain their own name as value.
>
   a[1,1] := 1: a[1,2] := 2: a[2,2] := 4:
# As with other kinds of tables, last-name evaluation instead of full
# evaluation holds.  Use print to view an array.
>
   a;
                                       a
>
   print(a);
array ( 1 .. 2, 1 .. 2,
                                    [1, 2]
```

```
                                [a[2, 1], 4)
# Symbolic indices create symbolic array references.  When the index
# expression is given a value, then full evaluation produces an actual
# array reference.
>
   i := 'i';
                                        i := i

>
   b := a[i+1,i];
                                  b := a[i + 1, i]

>
   c := 2*b;
                                 c := 2 a[i + 1, i]

>
   i := 1;
                                        i := 1

>
   b;
                                      a[2, 1]
# Below, a one-dimensional array with a subscript range -1 to 3 is created.
# In a manner similar to tables, we define that b[-1]'s value is t
# when we create the array.
>
   b := array( -1..3,[-1=t]);
b := array ( -1 .. 3,
                            [t, b[0], b[1], b[2], b[3]]
)

>
   b[-1] + b[1];
                                      t + b[1]
# Arrays (and tables) can have special indexing functions.  Creating a
# sparse array means that an array entry is zero unless explicitly set to
# something else.  Other built-in possibilities for indexing functions
# are symmetric and antisymmetric, but you can create your own.  See
# MRM or help(indexfcn) for further details.
>
   c := array(sparse, 1..2, 1..5, [(2,2)=5]);
c := array ( sparse, 1 .. 2, 1 .. 5,
                                  [0, 0, 0, 0, 0]
                                  [0, 5, 0, 0, 0]
)
```

4.12 Converting from one structure to another

The data structures used to represent sums, products, sets, lists, and arrays are highly similar. Each can be thought of as a collection of objects written down in some order. It's almost possible to think of transforming a sum to a list by replacing the + signs with commas and then putting brackets around the entire expression.

In some applications, it is desirable to do such a conversion. We can do this by means of the **convert** function. The command

convert(*expr*, *result type*)

will look in Maple's preprogrammed library of conversions to see whether there is preprogrammed knowledge of how to convert *expr* (a list, set, sum, product, etc.) to the desired data structure, one of the Maple type names in Table 8 or Table 9. See MRM or **help(convert)** for a complete list of target types. If such knowledge is in the library, the conversion happens according to its specifications. If the knowledge isn't in the library, then you will see the message **Error, unable to convert**.

Example 58

Converting between data structures

```
#Define a list named testscores.
>
   testscores := [99, 88, 76, 37, 92, 96, 88, 92];
                testscores := [99, 88, 76, 37, 92, 96, 88, 92]

#What is the average of these numbers?  Convert the list to a sum; the
#automatic simplifier will add them up.
>
   convert(testscores,`+`);
                                      668

>
   "/nops(testscores);
                                    167/2

#Use evalf for the "calculator" result.
>
   evalf(");
                                 83.50000000

#How many distinct scores are there?
>
   nops(convert(testscores,set));
                                       6

#Convert a list into an array/vector.
>
   a := convert([x,y,z],array);
a := array ( 1 .. 3,

                                   [x, y, z]

)

#Convert it back into a list.
>
   alist := convert(a,list);
                               alist := [x, y, z]

#Define a two-dimensional array with ones on the diagonal.
>
   A := array(1..3,1..3, identity);
                A := array ( identity, 1 .. 3, 1 .. 3, [] )

#Convert it into a list of lists; the i-th element of the list is
#row i of A.
>
   Alist := convert(A,list,list);
                 Alist := [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

4.13 The map function: Performing the same operation to all elements of a data structure

Maple built-in operations such as evalf, normal, and mod are defined for numbers or expressions but do the "obvious thing" when given a list or set as an argument: they return a list or set each element of which has had the built-in function applied to it. The function **map**(*proc*, *expr*) is useful in situations where you wish to do the same for *every* structure individually. The command

map(*function*, *expr*)

creates an expression where the i-th operand of the new structure is the result of applying *function* to the i-th operand of the original data structure. The type of the created expression is the same as that of the original data structure, unless the automatic simplification subsequently changes it. Thus, map works not only on lists and sets but also on sums, products, equations, and so on.

● ATTENTION

map also works on arrays and tables; but due to their "last-name evaluation" property, if A is an array or table, you have to say map(f,op(A)) instead of map(f,A).

Example 59
map applies a function to each component of a structure

```
#Effect of mapping evalf on a list, set, and sum
>
   L := [ 3/5, 47/95, ln(3)];
                                        47
                          L := [3/5, ------, ln(3)]
                                        95
>
   s := { ln(3), ln(4), ln(5)};
                          s := {ln(3), ln(4), ln(5)}
>
   expr := 1+ln(3)+exp(4);
                          expr := 1 + ln(3) + exp(4)
>
   map(evalf,L);
                  [.6000000000, .4947368421, 1.098612289]
>
   map(evalf,s);
                  {1.098612289, 1.386294361, 1.609437912}
#After mapping, Maple automatically simplifies the sum to a single
#floating-point number.
>
   map(evalf,expr);
                               56.69676232
```

```
#A little "advanced Maple" — define new functions f and g (see Chapter 5).
#f(x) is x squared.
>
   f := proc(x) x^2 end;
f := proc (x) x**2 end

#g(x) is the sequence x,x^2.
>
   g := proc (x) RETURN(x,x^2) end;
g := proc (x) RETURN(x,x**2) end

>
   L2 := [1,2,3,4];
                              L2 := [1, 2, 3, 4]

>
   s2 := {op(L2)};
                              s2 := {1, 2, 3, 4}

>
   A2 := convert(L2,array);
A2 := array ( 1 .. 4,

                                [1, 2, 3, 4]

)

#Map the operations of f and g onto the list L2 and set s2.
>
   map(f,L2);
                                [1, 4, 9, 16]

>
   map(g,L2);
                          [1, 1, 2, 4, 3, 9, 4, 16]

>
   map(g,s2);
                             {1, 2, 3, 4, 9, 16}

#"A2" does not evaluate to an array, but op(A2) does.
>
   map(f,A2);
                                      2
                                    A2

>
   map(f,op(A2));
array ( 1 .. 4,

                                [1, 4, 9, 16]

)
```

`map` can also be used in situations where you wish to apply a function of several arguments to all the terms of an expression *expr*, fixing all but the first argument at some predefined value. The command

map(*f*, *expr*, *arg2*, *arg3*, ..., *argn*)

creates a new data structure of the same type as *expr*, with *f(op(i, expr), arg2, arg3, ..., argn)* for the *i*-th operand.

Example 60
map with more than two arguments

```
#Compute the definite integral with respect to t from 1 to 2 for
#three expressions in a list.
>
   map( int, [exp(t), ln(t), tan(t)], t=1..2);
                                                    2
                                                   /
                                                  |
                [exp(2) - exp(1), - 1 + 2 ln(2),  |  tan(t) dt]
                                                  |
                                                 /
                                                1

#Compute the union of the solution sets of two equations.
>
   map(solve, {x^2-1=0, 5*x=4}, x);
                                  {-1, 1, 4/5}
```

The readlib-accessible function **zip**(f,u,v) takes two lists (or vectors) and produces a list (vector) whose i-th element is $f(u_i, v_i)$. See MRM or **help(zip);**.

4.14 Linear algebra in Maple; the `with` function; packages

The Maple library contains several *packages*, each package a collection of related functions. This section discusses one package, **linalg**, which is a collection of functions that work with the particular kinds of arrays used in Maple to represent vectors and matrices. Collectively, the `linalg` functions use the programming convention that a vector is an array of the form `array(1..n)` and a matrix is an array of the form `array(1..m, 1..n)`.

with(linalg) is the command to read in all the functions of the `linalg` package. Table 12 describes the names of common vector and matrix operations.

Example 61
The linear algebra package

```
# Define u to be a real 3-dimensional vector.
>
   u := array([-1,3-s,-4]);
u := array ( 1 .. 3,
                                [-1, 3 - s, -4]
)
# Define v to be a real 3-dimensional vector.
>
   v := array([1,t,1]);
v := array ( 1 .. 3,
                                   [1, t, 1]
)
```

TABLE 12
Matrix/vector operations

Operation	Maple notation
$u + v$ (vector)	add(u,v) or evalm(u+v)
$A + B$ (matrix)	add(A,B) or evalm(A+B)
AB	multiply(A,B) or evalm(A &* B)
Av	multiply(A,v) or evalm(A &* v)
$u \cdot v$	dotprod(u,v)
$u \times v$	crossprod(u,v)
A^{-1}	inverse(A) or evalm(1/A)
det A	det(A)
tr A	trace(A)
adj A	adjoint(A)
Solve $Ax = b$	linsolve(A,b)
Find the eigenvalues of A	eigenvals(A)
Number of components in a vector v	vectdim(v)
Number of columns in a matrix A	coldim(A)
Number of rows in a matrix A	rowdim(A)

This is only a partial list of the functions in `linalg`. MRM and **`help(linalg);`** supply more details about which linear algebra functions exist and how they work.

```
#Load in linear algebra package.
>
   with(linalg);
Warning: new definition for   trace
[crossprod, coldim, eigenvals, addcol, trace, rowdim, kernel, permanent, det,

   cond, laplacian, addrow, gausselim, smith, adj, mulcol, transpose, curl,

   adjoint, ismith, rowspace, mulrow, leastsqrs, mdet, angle, genmatrix,

   subvector, swapcol, norm, scalarmul, colspace, add, diverge, vandermonde,

   swaprow, hadamard, jacobian, range, definite, dotprod, submatrix,

   singularvals, linsolve, vectdim, sylvester, grad, band, indexfunc,

   nullspace, hessian, rank, multiply, orthog, hilbert, toeplitz, bezout,

   inverse]

>
   readlib(evalm):
>
   result := evalm(2*u+alpha*v);
result := array ( 1 .. 3,

                [- 2 + alpha, 6 - 2 s + t alpha, - 8 + alpha]

)
```

```
# Define A to be a 3 x 2 matrix.
#
>
   A := array([[1,2],[3,s],[4,8]]);
A := array ( 1 .. 3, 1 .. 2,
                                    [1, 2]
                                    [3, s]
                                    [4, 8]
)
#Solve A x = u.
>
   linsolve(A,u);
array ( 1 .. 2,
                                  [1, -1]
)
# vectdim describes the number of components in a vector.
# rowdim coldim describe the number of rows and
# columns in a matrix, respectively.
#
>
   vectdim(result);
                                     3
>
   rowdim(A);
                                     3
>
   coldim(A);
                                     2
#Vectors are not considered to be "column matrices" for multiplication,
#but are for the purposes of transposition.
>
   B := multiply(u,transpose(v));
Error, (in multiply) expecting a matrix
>
   B := multiply(transpose(transpose(u)),transpose(v));
B := array ( 1 .. 3, 1 .. 3,
                                [-1, - t, -1]
                            [3 - s, (3 - s) t, 3 - s]
                                [-4, - 4 t, -4]
)
#Compute the determinant of B.
>
   det(B);
                                     0
```

You can avoid reading in all the functions that go with the command `with(linalg)` by using the "long form" `linalg[`*funcname*`](...)`. For example, you can do `linalg[add](u,v)` at any point in a Maple session without doing a prior `with`.

Example 62
More linear algebra, without `with`

```
# Define I to be the 3 x 3 identity matrix.
>
   I := array(1..3,1..3, identity);
Error, may not assign to a system constant

#Oops, we forgot that "I" stands for the square root of -1.  Let's try again.
>
   IdentityMatrix := array(1..3,1..3, identity);
          IdentityMatrix := array ( identity, 1 .. 3, 1 .. 3, [] )

#The long form of a linalg function works even with no prior "with" command.
>
   linalg[add](IdentityMatrix,IdentityMatrix);
array ( 1 .. 3, 1 .. 3,
                                  [2, 0, 0]
                                  [0, 2, 0]
                                  [0, 0, 2]
)
```

● ATTENTION

When you use `with` to make access to a package's functions convenient, you occasionally will override a previous definition of a function, either built in or defined by you. For example, **trace** is by default defined to be the program-tracing function (see Section 5.15). However, after `with(linalg)`, `trace` is redefined to mean the trace of a matrix. The `with` function prints out a warning message whenever it overrides a previous definition.

In addition to `linalg`, there are packages for beginning calculus students (**student**), statistics (**stats**), power series manipulation, number theory, differential forms, and orthogonal polynomials, among others. The `with` function and the "long-form" style can be used with any of the other packages in a manner similar to that used with `linalg`. See MRM or **help(packages)** for further details.

CHAPTER 5

The Maple Programming Language

We have already seen many examples of two key elements of Maple's programming language, the *expression statement* (a command or expression) and the *assignment statement*. This chapter discusses the elements of more elaborate programming in Maple. It assumes that you are already familiar with another conventional programming language. Obviously, it is not intended to be an introduction to computer programming!

5.1 Repetition `while` you wait

`while` *condition*
`do`
  statement sequence
`od;`

causes repeated execution of *statement sequence* until the *condition*, an expression that is either `true` or `false`, is no longer `true`. The `while-do-od` do not have to be spread out over several lines, but sometimes are much more intelligible if laid out in this way and appropriately indented.

● ATTENTION

We have constructed the programming examples of this chapter as if the user had typed each line ending with enter. However, you may find it easier to follow the style for entering blocks of text suggested in Section 2.5.1 when you wish to do Maple programming.

Example 63
Use of `while` to compute integer gcd's: `printlevel`

```
# Example of repetition while a condition is true.
# irem(m,n) is the integer remainder of dividing the integer m by the
# integer n.
>
   irem(5,3);
                                2
```

```
>
   irem(12,3);
                                    0
# The following programs Euclid's algorithm for finding the greatest
# common divisor of two integers.  There is a Maple built-in function,
# igcd, which does this for you, but here's how you could get the
# computation by giving the directions yourself.
>
   a := 20;
                                 a := 20
>
   b:= 12;
                                 b := 12
>
   while irem(a,b)<>0 do
>
       d := irem(a,b);
>
       a := b;
>
       b := d;
>
   od;
                                  d := 8
                                 a := 12
                                  b := 8
                                  d := 4
                                  a := 8
                                  b := 4
>
   lprint(` integer gcd is`, d);
 integer gcd is   4
# You can suppress the printing of intermediate results in the repetition
# by assigning the variable printlevel to 0.  Don't forget to reset it to 1
# after you are done, though.  Printlevel is discussed further in Section 5.15,
# "Creating and debugging Maple programs."
>
   a := 35;
                                 a := 35
>
   b:= 15;
                                 b := 15
>
   printlevel:=0:
>
   while b<>0 do
>
       d := irem(a,b);
>
       a := b;
>
       b := d;
>
   od;
>
   printlevel:=1:
>
   lprint(` integer gcd is`, d);
 integer gcd is   5
```

5.2 Repetition for each one

To repeat a sequence of commands for each of a sequence of values (within a known range) assigned to an index variable i, you can use the following:

```
for i from start by change to finish
do
  statement sequence
od;
```

The statement sequence is repeated once for each distinct value of i.

Initially, the index variable i is set to the **start** value. If i is less than or equal to **finish** (greater than or equal to if **change** is a negative value), then the statement sequence is performed. Then i is assigned the value i + change. The process of comparison and execution of the statement sequence is then performed repeatedly until the value of i goes beyond the value of finish. The final value of i will be the i value that first exceeded finish.

● ATTENTION

A common oversight is to use the variable i in a for statement and then to use it unquoted in a sum or int. This, of course, causes the value of i left over from the for to be used in the sum or int, instead of the intended (unassigned) symbol i. Remember to use quotes or to unassign your for loop index variables after you're finished with their values.

start, change, and finish must be expressions that evaluate to rational or floating-point numbers. If the from start or by change is omitted, a default value of 1 is used for the omitted value.

Example 64
for statement

```
#Example of repetition with an automatically maintained index variable
>   for i from 2 by 2 to 6 do
>       lprint(`sum of `,j^i, ` j = 1..n is:`);
>       print( expand(sum(j^i,j=1..n)) );
>   od:
sum of    j**2    j = 1..n is:
                                3       2
                           1/3 n  + 1/2 n  + 1/6 n

sum of    j**4    j = 1..n is:
                                5       4       3
                      1/5 n  + 1/2 n  + 1/3 n  - 1/30 n
```

```
sum of    j**6    j = 1..n is:
                                  5        3        6        7
                  1/42 n + 1/2 n  - 1/6 n  + 1/2 n  + 1/7 n

#i's value after completion of the repetition is the next value it would
#take on if the repetition were to be continued.
>
   i;
                                      8
```

It is often useful to look at each part of an expression in turn and to use these parts in some computation. One way of doing this would be to write

```
for i to nops(expr)
do
  statements using op(i, expr)
od
```

—all quite legal, but not that attractive. This programming idiom was showing up so often in Maple programs that another form of the `for` was introduced:

```
for x in expr
do
  statements using x
od
```

Example 65
`for-in` statement

```
# Here we compute the sum of squares of even list members
# using both kinds of for loops.
>
   l := [1,2,3,4,5]:
>
   s := 0:
>
   for i to nops(l) do
>
       if irem(op(i,l),2)=0 then
>
           s := s + op(i,l)^2
>
       fi
>
   od:
>
   s;
                                     20

>
   s := 0:
>
   for n in l do if irem(n,2)=0 then s := s + n^2 fi od:
>
   s;
                                     20
```

The `for` and `while` statements can be combined. For example, if a rational-valued function `f` has been defined in the session, then

```
for i from 10 while f(i) < 100 do od;
```

would, after execution concludes, have as the value of `i` the smallest integer such that $i \geq 10$ and $f(i) \geq 100$.

5.3 Conditional execution with `if-then-else-fi`

If you wish *statement sequence 1* to be performed if *condition* is `true`, but *statement sequence 2* if it is not, then

`if` *condition* **`then`**
  statement sequence 1
`else`
  statement sequence 2
`fi;`

will do the trick.

Programming language etymologists will notice that Maple has a "do..od" and "if..fi" style of bracketing delimiters instead of "begin..end" or "if..endif". This style is attributable to Maple's having Algol 68 in its ancestry. The designers of Maple were influenced by other programming languages as well.

The *condition* referred to above can be a combination of equalities, inequalities, variables, or functions that `evalb` would evaluate to `true` or `false`, such as `a+b<6` or `divide(poly1,poly2)` and `lcoeff(poly1)=1`. Cases can be selected according to one of several conditions by

`if` *condition 1* **`then`**
  statement sequence 1
`elif` *condition 2* **`then`**
  statement sequence 2
`elif` *condition 3* **`then`**
  ⋮
`else` *condition n* **`then`**
  statement sequence n
`fi;`

Example 66
Using `if` to print out results of primality testing

```
#isprime(n) is a function automatically loaded from the Maple library that
#returns true if n is prime, false otherwise.
>
   isprime(37);
                                  true
```

```
>
   isprime(6);
                                    false
#Inspect the numbers of the form 2**(2*n+1)-1.
#Set printlevel to 0 to display only the results of print.
>
   printlevel:=0:
>
   for i to 8 do
>
     prime_candidate := 2**(2*i+1)-1;
>
     if isprime(prime_candidate)
>
     then print(prime_candidate, ` is prime`);
>
     else print(prime_candidate, ` is not prime`)
>
     fi;
>
   od;
                                7,  is prime
                               31,  is prime
                               127,  is prime
                              511,  is not prime
                             2047,  is not prime
                              8191,  is prime
                             32767,  is not prime
                              131071,  is prime
>
   printlevel:=1:
```

Example 67

if ... then ... elif ... fi

```
#Search for formulae whose integral from 0 to 1 is about 1.0.
>
   candidate_list := [.8*sin(x), exp(x), 3.1*x^2, exp(.7*x)];
                                                          2
           candidate_list := [.8 sin(x), exp(x), 3.1 x , exp(.7 x)]
# L[i] is the i-th element of a list L.
>
   printlevel:=0:
>
   for candidate in candidate_list do
>
       area := evalf(int(candidate,x=0..1));
>
       if area > 1.1
>
       then print(candidate, ` too high `);
>
       elif area < .9
>
       then print(candidate, ` too low`);
```

```
>
      else print(candidate, ` acceptable`)
>
      fi;
>
   od;
                              .8 sin(x),  too low
                               exp(x),  too high
                                    2
                              3.1 x ,  acceptable
                             exp(.7 x),  too high
>
   printlevel:=1:
```

5.4 break and next: Control within for-while loops

The value **break**, when encountered during a `for-while` repetition, causes immediate exit from the (innermost) `for-while` loop containing the `break`. An example of its use is in "search" situations, when you would like to stop looking through a series of possibilities as soon as you find what you are looking for.

When the value **next** is encountered during a `for-while` repetition, all statements are skipped from that point to the bracketing `od` of the (innermost) repetition loop. Execution then proceeds to the consideration of whether to perform another iteration.

Example 68
Leaving a `for` repetition early with `break`

```
# Find the first non-prime in the sequence 2^i-1, i=3,5,7,...
# When you hit a non-prime, break out of the repetition.
# isprime(n) is a built-in function that is true if n is a prime number,
# false otherwise.  See MRM or help(isprime) for further details about it.
>
   for i from 3 by 2 do
>
      if isprime(2^i-1)
>
      then print(2^i-1, ` is prime`)
>
      else break
>
      fi;
>
   od;
                                 7,  is prime
                                31,  is prime
                                127,  is prime

#The value of i after the break is its value when the break occurred.
>
   print(`i=`,i, ` first non-prime in the sequence is`, 2^i-1);
                 i = 9,  first non-prime in the sequence is,  511
```

5.5 Simple Maple procedures

Users can define their own functions by using procedure (**proc**) expressions. All Maple procedures are functions, in that they are given values or names of objects as arguments and return a value as a result. Procedures, like arrays, lists, and mathematical expressions, are objects that are considered valid values for any programming variable to be assigned. Thus, a programming variable can be assigned a `proc` expression as its object. The name of the variable is the name of the function. Using a function defined through `proc` is the same as using any other function, with the "name of the function" being the name of the programming variable.

The rules for evaluation of a `proc` work the same as for tables. If a variable `f` has a `proc` as its value, the value of `f` displayed is the name of the `proc` in the sense of *last-name evaluation*, just as discussed in Section 4.11.1. If you wish to get at the `proc` object instead of the function name, you must use `op(f)`. However, this does not interfere with being able to use a procedure—for example, `f(3,5)`—in the natural way.

The simplest form of a Maple function definition consists of assigning a programming variable something of the form

proc (*sequence of formal parameters*)
  proc's expression involving the parameter sequence
end

The *sequence of formal parameters* in a procedure definition must be a sequence of names. Within the `proc-end`, whatever values these symbols have outside the `proc` is ignored; they are used just as placeholders in describing what happens later when a computation is requested.[1]

To ask for a computation involving the function ("call the function"), include in a command, or as part of a statement, the name of the function (the programming variable assigned the `proc` definition as a value), followed by a sequence of *arguments* enclosed by parentheses:

`fname(`*`expr1`*`, `*`expr2`*`, ...)`

If the arguments involve programming variables as well as symbols or numbers, then "full evaluation" is applied to get values to be given to the function. Those values are then substituted into the `proc`'s *expression*. The `proc`'s *expression* is then evaluated and simplified. The result of evaluating and simplifying the expression is the value of the function.

[1] Thus, procedure/function definition is another exception to the policy of "full evaluation."

Example 69
A simple Maple function defined and used

```
# Define a function/procedure named f.  Below, the sequence of formal
# parameters is x, y.
>
   f := proc ( x , y )
>
     x^2-y^2;
>
        end;
f := proc (x, y) x**2-y**2 end

# Evaluate the function f with the arguments 3, 5 (i.e., x=3, y=5).
>
   f(3,5);
                                    -16

# Evaluate the function f at x=w, y=z+1.
>
   f(w,z+1);
                                 2         2
                               w  - (z + 1)

# An example of "last-name evaluation" — the value is not the
# proc, but the last name in the chain of values.
>
   g:= f;
                                   g := f

>
   h:= g;
                                   h := f

>
   h(3,5);
                                    -16

>
   f;
                                     f

>
   g;
                                     f

>
   h;
                                     f

# As with tables, op or print can be used to view the proc object.
>
   op(f);
proc (x, y) x**2-y**2 end

>
   print(h);
proc (x, y) x**2-y**2 end
```

Conceptually, built-in commands such as `diff` or `solve` are like mathematical functions: given arguments, they return one value (expression) as a result. However, procedures can have effects beyond the values they return. For example, the Maple function `divide` has as an additional effect the assignment of a value to the programming variable given as its optional third argument. Another kind of side-effect would be a function that prints a "divide by zero" error message. Additionally, programming

variables not included amongst the sequence of formal parameters may influence its results, as in the case of the effect of `Digits` on `evalf`.

5.6 More Maple procedures: Multiple statements, local variables, RETURN

Sometimes it is not convenient to define the result of a procedure in terms of only one expression, as in the preceding section. The general form of a Maple `proc` definition is

proc(*parameter sequence*)
  local *variable sequence*;
  options *option sequence*;
  statement 1;
  statement 2;
    ⋮
  statement n
end;

The `local` and `options` sequences can be omitted if you don't need them. We will discuss `options` in the next section and `local` later on in this one.

The semicolon of the last statement in a `proc` (as well as in an `if` or a `for-while` body) is optional. According to the syntax rules of Maple, semicolons are marks that separate statements but are not considered mandatory to punctuate the end of a statement sequence.

When Maple procedures involve several statements in their definition, the statements are performed one at a time, with the last value computed being the value of the procedure.

Example 70
A simple multistatement procedure defined and used

```
#A "toy" function for finding the maximum of three numbers
#There is actually a Maple library function "max" that you could use
#instead.
>
   max3 := proc (a,b,c);
>
     print(`Finding the maximum of`,a,b,c);
>
     if a<b
>
     then if b<c
>
        then c
>
        else b
>
        fi
```

```
>
    elif a<c
>
       then c
>
       else a
>
    fi;
>
  end:
#max3(3,2,1); executes the statements in the proc body with a having
#value 3, b having value 2, c having value 1.
>
  max3(3,2,1);
                        Finding the maximum of, 3, 2, 1
                                      3
>
  max3(1,1,9);
                        Finding the maximum of, 1, 1, 9
                                      9
```

Local variables are programming variables that are used only during execution of the `proc` and then discarded. If a local variable has the same name as a variable known outside the procedure, then when the name occurs in the statements defining the procedure, it refers to the local variable, not the exterior one(s).

Local variables are evaluated using *one-level evaluation*. This means that when a local variable is evaluated, its value is the same as the last time it was assigned; its value is *not* reevaluated to see whether more simplifications can be done. Such evaluation of "locals" is normally what one wants when programming, and from a practical point of view it leads to more efficient programs. If for some reason you want to fully (re-)evaluate a local variable, then you can use the `eval` function.

Example 71
Evaluation of local variables

```
# Evaluation of global names at top level uses full evaluation.
# The second evaluation of A sees that w is zero.
>
  A := sin(w): A; w := 0: A;
                                    sin(w)
                                      0
# Evaluation of global names in a proc is the same.
# It uses full evaluation too.
>
  Bfun := proc() B := sin(x); print(B); x := 0; print(B) end:
>
  Bfun();
                                    sin(x)
                                      0
```

```
# Evaluation of local names is different: one-level evaluation is used.
# This means that the second evaluation of C will not see that y is zero.
>
   Cfun := proc() local C; C := sin(y); print(C); y := 0; print(C) end:
>
   Cfun();
                                    sin(y)
                                    sin(y)
# We can, however, use "eval" to fully evaluate a local variable.
>
   Dfun := proc() local D; D := sin(z); print(D); z := 0; print(eval(D)) end:
>
   Dfun();
                                    sin(z)
                                      0
```

Within a proc, you can retrieve the number of actual arguments specified through the special name **nargs**. The special name **args** has as its value the sequence of actual arguments. The i-th argument in the sequence args is obtainable through the selection construct args[i]. This makes it possible to write procedures that are flexible in the number of arguments they take.[2]

Example 72

Procedure using local variables

```
# The example below is a procedure to find the maximum of all the
# numbers given as arguments.  Any number of numbers can be supplied.
# All parameters are referred to via args[i], for some value of i. Thus,
# there is no need to describe the sequence of parameters, just that
# there is one.
>
   maxN := proc ()
>
     local result, i;
>
     if nargs>0
>
     then
>
         result := args[1];
>
         for i from 2 to nargs do
>
             if args[i] > result then result := args[i] fi;
>
         od;
>
         result;
>
     fi;
>
   end:
>
   maxN(25/7, 525/149);
                                    25/7
>
   maxN( 25/7, 525/149, 9/2);
                                    9/2
```

[2] nargs and args are not *variables*, because they cannot be assigned values in a procedure.

```
#A demonstration for computer scientists and computer science students
#that Maple does not have nested scopes
>
  h := proc (x)
>
    local w;
>
    w := x + 1;
>
    i(x);
>
  end:
>
  i := proc (a) a-w end:
>
  h(5);
                                  5 - w
#The reference in i is to the global (top-level) variable w, not h's
#local variable of the same name.
>
  w := 47;
                                 w := 47
>
  h(5);
                                  -42
```

5.7 `Option remember`: Remembering previously computed function values

Among the *options* recognized by Maple in version 4.2 is **`option remember`**.[3] Whenever the value of a procedure with `option remember` is computed, the Maple system makes an entry in a table "remembering" the value of the procedure for the given arguments. When the procedure is called again with the same arguments, the Maple system retrieves the result from the procedure's *remember table*. In this way, it is possible to avoid redundant computations. For instance, in Example 2, which defined the Fibonacci sequence 1, 1, 2, 3, 5, 8, ..., `option remember` is used. For any integer k, computing the k-th Fibonacci number F_k requires knowing the values of F_{k-1}, and F_{k-2}, which in turn requires knowing $F_{k-3}, \ldots, F_1$. The way the procedure works in Example 2, all values of F_1 through F_{k-1} are computed and remembered by the time the value of F_k is computed. If later in the session F_{k+1} is ever computed, Maple will access the remembered values of F_k and F_{k-1}. If F_{100} were computed without `option remember`, over 3×10^{20} `fibonacci` function calls would be required, most of them redundant. With `option remember`, only 100 `fibonacci` computations would be needed.

One can force the remembering of some special values by assignment. If the procedure `func` has already been defined (with or without `option remember`), then the assignment statement

`func(`*argument sequence*`) :=` *value*`;`

[3] The other options are **`builtin`**, **`operator`**, **`trace`**, and **`system`**. See MRM or **`help('option')`** for further details on them.

stores in the procedure's *remember table* the "fact" that the value of `func(argument sequence)` is *value*. The left-hand side of the assignment is not fully evaluated; the function name is evaluated and the arguments are evaluated, but the function is not called.

Example 75
A function remembering a special value

```
#Example of use of remembering of a special value
>
   f := proc(x) sin(x)/x end:
>
   f(0) := 1;
                                  f(0) := 1

>
   f(infinity) := 0;
                                f(infinity) := 0

>
   [f(0), f(1), f(Pi/2), f(Pi), f(infinity)];
                          [1, sin(1), 2 1/Pi, 0, 0]
```

5.8 Using ERROR: Exiting several procedures at once

To indicate that something has gone wrong in a procedure you have defined, you can use the **ERROR** function. The general form is

ERROR(*error message*)

(All the letters in ERROR must be capitalized, as with RETURN.)

Executing an ERROR statement in a procedure prints a message of the form

`Error, (in functname)` *error message*.

As with all errors, the computation is interrupted.

Example 76
Using the ERROR statement to signal a computational exception

```
#Finding the maximum of some numbers as before, but this time an
#error occurs if no numbers are provided.
>
   maxN := proc ()
>
     local result, i;
>
     if nargs=0 then ERROR(`no arguments to maxN`) fi;
>
     result := args[1];
>
     for i from 2 to nargs do
>
        if args[i] > result then result := args[i] fi;
>
     od;
```

```
# max with no arguments returns the null sequence, since there was no
# "last value computed" in the function.
>
   maxN();
>
   maxN(25/7);
                                    25/7
# The local variable "result" has no assigned value outside of the
# procedure.  In fact "maxN's local variable result" is not even
# the same object as the "result" symbol below.
>
   result;
                                  result
```

Occasionally, when defining a multistatement procedure, you may find that the required function value is already in hand after only some of the specified actions have been performed. It is possible to return the value at that point via

RETURN (*value*) .

● ATTENTION

RETURN is different from the names return or Return. Spell it in all uppercase letters.

Example 73 .
A procedure to compute Chebyshev polynomials using RETURN

```
#Chebyshev(n):  A procedure to compute the Chebyshev
#polynomials of degrees 0 through n.  A table is returned
#as the value of the procedure.
#The first two Chebyshev polynomials are defined explicitly.  If
#"Chebyshev" is ever called with n=0 or n=1, those polynomials are
#returned with no further computation.  For larger values of n, more
#computation is necessary —the k-th polynomial is defined in terms of
#the previous two.  Note that this function has as its value a table —
#many expressions are being returned at once.
>
   Chebyshev := proc (n)
>
      local p,k;
>
      p[0] := 1; p[1]:=x;
>
      if n<=1 then RETURN( op(p) ) fi;
>
      for k from 2 to n do
>
         p[k] := expand( 2*x*p[k-1] - p[k-2] )
>
      od;
>
      RETURN( op(p) )
>
   end:
# An example using the Chebyshev procedure.
>
   a := Chebyshev(5):
```

```
# a now has as its value a table with six entries.
>
  a[i] $ i = 0..5;
                                  2          3          4       2          5       3
      1, x, 2 x  - 1, 4 x  - 3 x, 8 x  - 8 x  + 1, 16 x  - 20 x  + 5 x

#Display the value of the first six Chebyshev polynomials at x=1/4.
>
  x := 1/4:
>
  a[i] $ i = 0..5;
                                                  11   17   61
                        1, 1/4, -7/8, - --, --, --
                                                  16   32   64
```

As we have mentioned, if you have a local variable x within a proc, then x within the proc has no connection with the usage of x outside the proc. A consequence of this feature of Maple is that local names can be used as mathematical symbols as well as programming variables, but although a global variable and local variable may have the same name, and may be printed out the same, they are not the same (equal)!

Suppose the procedure f calls or defines the procedure g within it. A statement within g cannot directly refer to any local variable of f (or vice versa, of course). A reference to a symbol x in g refers either to g's local variable x, if it has one, or the "global" symbol x (the x that is defined at the interactive level, before or after any Maple procedures are used).

Example 74

The meaning of names when one procedure is defined within another

```
#The reference in g below is to the global (top-level)
#variable y, not f's local variable of the same name.
>
  f := proc (x)
>
    local y;
>
    y := x + 1;
>
    g := proc (a) a-y end;
>
    g(x);
>
  end:
>
  f(5);
                                    5 - y
>
  y := 47;
                                  y := 47
>
  f(5);
                                   -42
```

```
>
    result;
>
  end:
>
  maxN( 25/7, 525/149, 9/2);
                                    9/2

>
  maxN();
Error, (in maxN) no arguments to maxN
```

The **traperror** function allows programmers to avoid this interrupt/abort behavior in programs they write. As its name suggests, traperror is used to catch an error that is in the process of happening. It is typically used when you have reason to believe that a computation may cause an error to occur *and* you know what to do if one does. The variable **lasterror** records the nature of the last ERROR and may be used to control what the program does.

Example 77

lasterror and traperror

```
# This computation normally causes execution to be interrupted.
>
  (1/3) mod 12;
Error, minvs: the modular inverse does not exist

# The fact that this particular error occurred is recorded in lasterror.
>
  lasterror;
                minvs: the modular inverse does not exist

# We can wrap the computations with traperror.  This will give
# the inverse or the string of the error message.
>
  u := traperror((1/4) mod 12);
              u := minvs: the modular inverse does not exist

>
  v := traperror((1/5) mod 12);
                                  v := 5
```

Example 78

Trapping an expected error

```
# The following procedure shows a trick one could use in computing limits.
# It simply evaluates the expression at the point and traps any errors that occur.
>
  try := proc(expr, var, val)
>
     exprval := traperror(normal(subs(var = val, expr)));
>
     if exprval <> lasterror then exprval else FAIL fi;
>
  end:

>
  h := (x-2)*(x+2)/(x**2 - 1);
                                  (x - 2) (x + 2)
                          h := ------------------
                                        2
                                       x  - 1
```

```
> try(h, x, 0);
                                4

> try(h, x, 1);
                               FAIL
```

Now we have seen how to indicate an error and how to trap one. It is also interesting to use ERROR and traperror in conjunction to cleanly escape from a computation. For example, in computing an integer factorization, if a factor is discovered deep inside a subprogram, you may want to return it immediately. (Some other programming environments, including C-based and Lisp-based ones, allow this sort of nonlocal return.)

Example 79

Nonlocal returns using ERROR/traperror

```
# Use ERROR to throw a value to an awaiting traperror.
> THROW := ERROR:
# Use traperror to catch the expected value.
> CATCH :=
> proc(tag, expr)
>    local x;
>    x := [traperror(eval(expr))];
>    if op(1,x)<>tag then ERROR(op(x)) fi;
>    op(2,x)
> end:
# Now implement a tail recursive version of factorial to try it.
> fact := proc(n) CATCH(`Factorial Catch`, 'factmult(n, 1)') end:
> factmult :=
> proc(n, prod)
>   if n = 1 then THROW(`Factorial Catch`, prod) fi;
>   if not type(n, integer) then ERROR(`An integer is required`) fi;
>
>   factmult(n-1, n*prod)
> end:
> fact(6);
                               720

> fact(x);
Error, (in CATCH) An integer is required
```

5.9 Packages in Maple

We have already discussed some of the packages available in Maple. Here we describe how to go about writing a new package.

A package is simply a table in which the entries are procedures. We see that the "long form" of names for the things in the package are really just normal table references. For example, `simplex` is a package. `minimize` is a key so `simplex[minimize]` is an entry from the table. Then `simplex[minimize](f,C)` calls the function entry in the table.

A call to `with` reads a package in from the library and performs a number of assignments to make the functions available via their normal names. When the command `with(simplex);` is given, an assignment of the form

```
minimize := simplex[minimize];
```

is executed, along with many others.

With this knowledge, it becomes fairly easy to write your own package.

Example 80
A simple package

```
# A package implementing double-ended queues

dequeue[dq] := proc()
    local t;
    t := table();
    t['maxix'] := -1;
    t['minix'] := 0;
    op(t)
end:

dequeue[putfront] := proc(t,x)
    local M;
    M := t['maxix'];
    t['maxix'] := M + 1;
    t[M+1] := x
end:

dequeue[putback] := proc(t,x)
    local m;
    m := t['minix'];
    t['minix'] := m - 1;
    t[m-1] := x
end:

dequeue[takefront] := proc(t)
    local M;
    if t['minix'] > t['maxix'] then ERROR(`dequeue is empty`) fi;
    M := t['maxix'];
    e := t[M];
    t['maxix'] := M-1;
    t[M] := evaln(t[M]);
    e
end:
```

```
dequeue[takeback] := proc(t)
    local  m;
    if t['minix'] > t['maxix'] then ERROR(`dequeue is empty`) fi;
    m := t['minix'];
    e := t[m];
    t['minix'] := m+1;
    t[m] := evaln(t[m]);
    e
end:

save `dequeue.m`;
```

As far as long names are concerned, using this package is the same as using any other. You need to read in the package first, though, if you are in a new Maple session.

Example 81
Using your own package

```
>
   read `dequeue.m`;
>
   d := dequeue[dq]():
>
   dequeue[putfront](d, 100):
                                    100
>
   dequeue[putfront](d, 200);
                                    200
>
   dequeue[takefront](d);
                                    200
>
   dequeue[putfront](d, 999):
                                    999
>
   dequeue[takeback](d);
                                    100
```

5.10 Your Maple initialization file

If you have just one or two packages, you can read them in whenever you want to use them. If it turns out that you load them almost every time you use Maple, then you might like to have them read automatically when Maple starts up. You can accomplish this with a *Maple initialization file*.

In fact, you can have Maple execute any sequence of commands you like every time you start a new session by creating a file of commands. This file is read after Maple has finished getting itself ready but before it starts to ask you for input.

What you put in the file depends on what you want to do. If you want Maple to be particularly enthusiastic, then you can put commands such as these in your initialization file:

```
print(`Welcome back! `):
print(`It's time for mooooore Maple! `):
prompt := `More Maple! `:
```

Notice that all the preceding commands ended in colons. It is a common practice to end all commands in a Maple initialization file with colons to keep the results of the commands from printing. In our case, of course, the commands were calls to the `print` function, so it didn't really matter.

The initialization file needs to be given a particular name in order for Maple to find it. Maple first looks for the text file `MapleInit` in the Maple folder and then searches your active system folder.

Getting back to packages, if the only package you had was `dequeue`, then you could have a Maple initialization file that contained

```
read `MyDisk:MapleStuff:dequeue.m`:
```

to read your package into every new session.

5.11 File Names

5.11.1 Folder name

We have seen various examples of the use of file names in illustrations given of the `read` and `save` commands. In the hierarchical file system of the Macintosh, which folder would contain these files? If you don't prefix the file name with an explicit path name, then the *default folder* is used. The default folder is the one containing the copy of Maple that you launched to start the session.

Alternatively, you can use *explicit path names* for any Maple commands that take a file name. A colon (:) is used to separate folder names. For example, the command

```
save `Engineering:Beam Analysis:Stress Results.m`;
```

would save the results of the session in a Maple internal-format file named `Stress Results.m` which is contained in the folder `Beam Analysis` on the disk volume named `Engineering`.

A path name that *begins* with a colon refers to a subfolder within the default folder. For example, the Maple variable `libname` contains the name of the folder that holds the Maple library. Ordinarily it has the value `:Lib`, since the Maple library was originally installed by you in the same folder (the default folder) as the Maple program.

5.11.2 Name translation

Many of the file names used in the Maple library contain words separated by the slash character (/). An example is `ifact/src/wheelfact`. These names translate directly into path names on many computer systems, such as those using Unix, where the slash character separates directory and file names. These names are retained in the library on the Macintosh, but before a file name is used, the slashes are translated into colons, the punctuation Macintosh uses to separate volume, folder, and file names.

Here's an example:

```
readlib (`ifactor/wheelfact`);
```

The `readlib` routine reads a Maple binary file from the library and executes the following statement:

```
read ``.libname.`/ifactor/wheelfact.m`;
```

After concatenation of the various components it becomes

```
read `:Lib/ifactor/wheelfact.m`;
```

assuming that `libname` has the value `:Lib`. Before the concatenated name is used to actually open a file, it is converted into a Macintosh file name

```
`:Lib:ifactor:wheelfact.m`
```

This `wheelfact.m` file is expected to be in the `ifactor` folder which is inside the `Lib` folder which is inside the default folder.

When you specify a sample name, you can use either a slash or a colon to separate volume, folder, and file names. Experienced Macintosh users probably prefer to use colons exclusively.

Note that file names on the Macintosh are case insensitive. The two commands

```
read `HD 80:Engineering:Beam Analysis:Stress.m`;
read `hd 80:engineering:beam analysis:stress.m`;
```

refer to the same file.

The Maple function **convert(...,hostfile)** converts a Maple-style filename into the Macintosh format.

5.12 Creating your own library

Once you have a number of your own packages, you will *not* want to read them all into every Maple session. You will want to start creating your own library.

To use your library, you have to tell Maple where to find it. (It is not a good idea to add files directly to the built-in "Maple" library. That makes it harder to install new releases of Maple, and you might overwrite something that really has to be there.) The `with` command normally looks in the Maple library for packages. This location is specified by the global variable `libname`, as described in Section 3.12. If you have your own library, you can get `with` to look there as well by means of the variable `_liblist`. (The first character in the name is an underscore.) You can either assign to this variable in every session or you can do it in your Maple initialization file. Your Maple initialization file might contain:

```
mylibname:= `MyDisk:MyMapleLib`:
_liblist := [mylibname]:
```

In general, the `_liblist` variable may refer to a list of libraries. If you share your computer with other people, you might want to have one private library and one common library. Then your initialization file would contain something like the following:

```
mylibname:= `MyDisk:MyMapleLib`:
sharedlibname := `DiskServer:MapleLib`:
_liblist := [mylibname,sharedlibname]:
```

Now that you have told Maple where to find your library, you need to put your packages in it. The simplest way to do so is to move your ".m" files to the place specified by `_liblist`. This would mean putting your ".m" files in the named `folder`. In general, the place for the ".m" file for the package `mypack` is given by the result of the Maple command

```
convert(`` . mylibname . `/mypack.m`, hostfile);
```

and the place for the source file is

```
convert(`` . mylibname . `/src/mypack`, hostfile);
```

Once you have done this, you can use `with` to load your packages and make the functions available by their short names.

Example 82
With a package from your library

```
# Assume that the .mapleinit file has the following assignments
# mylibname:=`:MyMapleLib`:
# _liblist := [`:MyMapleLib`,``]:
# Once dequeue.m has been added to the MyMapleLib folder, it becomes
# accessible via "with".
with(dequeue);
                [putback, takeback, takefront, dq, putfront]
```

```
d := dq():
putfront(d, thing1);
                                thing1
putfront(d, thing2);
                                thing2
takeback(d);
                                thing1
```

If a package is large, you may not want to load it all at once. To load the functions of a package one by one, as they are needed, you give `readlib` definitions for the individual entries.

Example 83
Setting up a `readlib`-ed package

```
# File ``.mylibname.`:src:deq2`
# Rather than defining the procedures directly in this file,
# we only give readlib definitions.
>
   deq2 := 'deq2':
>
   deq2[dq] :=
>
     'readlib(``deq2/dq``,       ``.mylibname.`:deq2:dq.m`)':
>
   deq2[putfront] :=
>
     'readlib(``deq2/putfront``, ``.mylibname.`:deq2:putfront.m`)':
>
   deq2[putback] :=
>
     'readlib(``deq2/putback``,  ``.mylibname.`:deq2:putback.m`)':
>
   deq2[takefront] :=
>
     'readlib(``deq2/takefront``,``.mylibname.`:deq2:takefront.m`)':
>
   deq2[takeback] :=
>
     'readlib(``deq2/takeback``, ``.mylibname.`:deq2:takeback.m`)':
>
   save deq2, ``.mylibname.`:deq2.m`;
```

You also need to create one ".m" file for each function in the package.

Example 84
A member of a `readlib`-ed package

```
# File ``.mylibname.`:deq2:src:dq`
`deq2/dq` := proc()
    local t;
    t := table();
    t['maxix'] := -1;
```

```
    t['minix'] := 0;
    op(t)
end:
save `deq2/dq`, ``.mylibname.`:deq2:dq.m`;
```

With this definition, all you need to do is to type `with(deq2);` and proceed as before. If you trace `readlib`, you will see that it loads the functions when they are first called.

5.13 Monitoring time and space consumed during a computation

Maple's way of showing you the progress of your computation is the **Status** window (see Figure 12). The window displays the number of bytes requested for data as the session progresses. As any computation proceeds, the amount of memory requested for new temporary results grows monotonically. Watching the number of bytes used will give you an indication of which expressions or commands demand a large amount of memory for their computation.

As further explained in Section 5.14, computer memory can be recycled. Thus, while the number of "bytes used" always increases during a session, this does not mean that there are that many *distinct* bytes of memory in use; some storage may

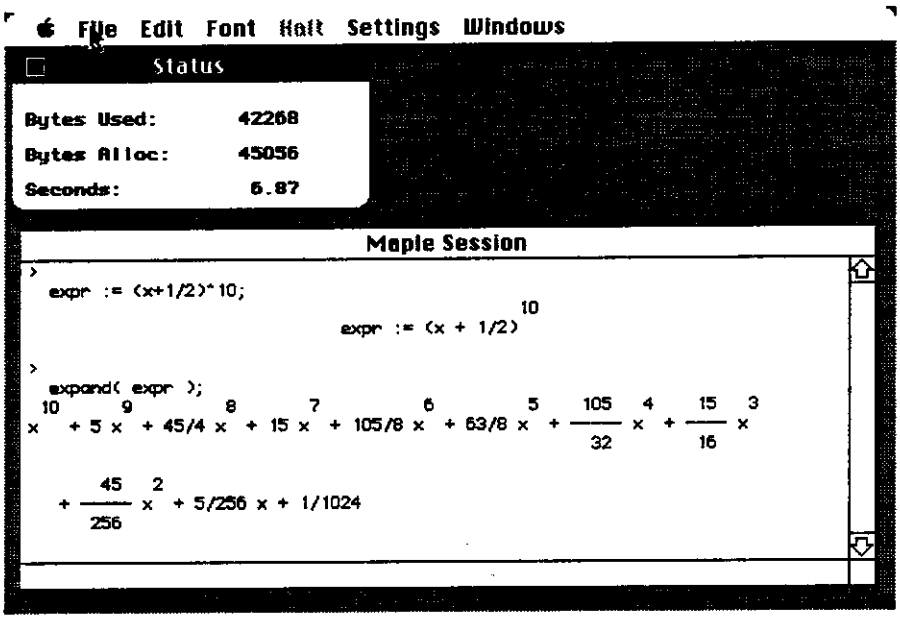

File Edit Font Halt Settings Windows
Status
Bytes Used: 42268
Bytes Alloc: 45056
Seconds: 6.87
Maple Session
> expr := (x+1/2)^10;
expr := (x + 1/2)^10
> expand(expr);
x^10 + 5 x^9 + 45/4 x^8 + 15 x^7 + 105/8 x^6 + 63/8 x^5 + 105/32 x^4 + 15/16 x^3
+ 45/256 x^2 + 5/256 x + 1/1024

FIGURE 12. The **Status** window

have been reclaimed from previous usage. In fact, if few results are kept in variables from one computation to the next and if garbage collection (or memory reclamation) is efficient, then the "bytes allocated" will grow very slowly, or not at all, while "bytes used" grows in accordance with new temporary data structures that are created, and perhaps later destroyed.

The time shown in the **Status** window is the number of seconds used for computation since the beginning of the Maple session.

To watch the memory consumption of a computation, you need to make the **Status** window visible during the computation. There are several ways of doing this. One is to resize all windows on the screen so that the **Status** window is never blocked by another window. Another way is to leave the **Status** window mostly hidden by other windows, but to bring it to the front after a computation has started by clicking on the **Status** window after pressing enter. Finally, you can use the **Window** menu to select the **Status** window and bring it to the front.

To take the **Status** window off the screen entirely, click in its go-away box in the upper left corner of the window. Even after the window has gone away, you can bring it back to the front of the screen by selecting it on the **Window** menu.

As any Maple user can easily observe, a complicated Maple computation may take a noticeable amount of time. Symbolic computation can often require a lot of computer memory as well. Computer memory requirements are dictated not only by the length of the answer (e.g., the number of terms, symbols, or digits of numbers), but also by the size of the expressions generated in intermediate subcomputations along the way. The **time** and **words** functions can help you monitor how these resources are being consumed.

time()
words(*integer*)
words()

The `time` function can be used to measure how long a Maple computation takes. The value of `time()` (`time` is a function with no arguments) is how many seconds the Macintosh has spent on Maple computations since the Maple session was started. Thus, taking the difference of two calls to `time` will produce the time elapsed between them.

Example 85

Computing elapsed time with the `time` function

```
#Timing a Maple computation
>
   time1:=time();
                                time1 := .483
>
   factor(x*y^2-x);
                              x (y - 1) (y + 1)
>
   time2:=time();
                                time2 := 2.433
```

```
#Time of factor computation, in seconds
> time2-time1;
                                1.950
```

The **showtime** function is a way of printing out time (and space) information automatically on every Maple command. See MRM or **help(showtime)** for further details.

The amount of computer memory used for Maple computations can also be measured in *words*, where a word is 4 bytes. By default, the **Status** window is updated every 25,000 words requested. The design of Maple allows a check for this condition only when it is convenient. Thus, when a large chunk of memory is used quickly (e.g., when a large procedure is read in from the library), you may see larger gaps between **Status** window updates.

The command `words(`*n*`)` changes the frequency of **Status** window updates to occur after every *n* words requested. `words(500000);` will reduce the frequency to be every 500,000 words. `words(1000000);` reduces the frequency even more, while `words(0)` will turn off **Status** window updates. You can change the `words` setting as many times as you wish during a session.

The value of `words();` is the total number of words used so far in the session. You can use `words();` if you wish to display the amount of memory used in the **Maple Session** window or want to use the quantity in a Maple computation or program.

5.14 Garbage collection and gc

gc(*integer***)** or
gc()

Some of the expressions generated during a computation are "scratch work" expressions. Although they are necessary, they are never displayed to the user and no longer need to be retained once the eventual result is produced. The memory used for these intermediate expressions can be reclaimed and recycled for later computations in the session through a process known as *garbage collection*. Garbage collection is important for larger computations, because your computer has a limited amount of memory, and it is not unheard of for a symbolic computation to run through several million words of storage.

During a garbage collection, storage no longer being used is reclaimed. Most of the computer memory used by Maple must be scanned to determine what must be retained and what can be recycled. For longer computations, the memory to be scanned can amount to several million words of memory. Thus, garbage collections can lead to a noticeable pause in the progress of the computation, from one to several seconds.

The `gc` function can be used to change the frequency and occurrence of garbage collections. `gc();` will cause a garbage collection immediately, whatever the automatic setting is. The only visible result of the `gc` function is a printed message summarizing the words used, words allocated, and time consumed, identical to the one

generated by `words`. `gc(500000);` will decrease the frequency of garbage collection to increments of 500,000 words used. In general, `gc(`*integer*`)` will cause an immediate garbage collection and change the frequency of garbage collection so that it occurs with every *integer* increments of words used.

The appropriate frequency of garbage collection depends upon how much memory is available on your Macintosh, whether you are using Multifinder and other programs concurrently with Maple, and the size of problems you are solving. On configurations where the amount of memory available to Maple itself is small, use garbage collection frequently to reclaim storage. On a Macintosh with many megabytes of memory, performance is not improved significantly by frequent garbage collection.

The default setting for Maple is 25,000, that is, garbage collection is performed automatically after every 25,000 words (or 100,000 bytes) are used. While the default setting is appropriate when using one megabyte of memory, `gc(250000)` is usually more appropriate for larger amounts of memory, such as four megabytes.

5.15 Debugging Maple programs: `printlevel` and `trace`

Computer users each have their own methods for creating an algorithm for solving a problem and verifying that it is correct. It certainly isn't the point of this section to teach you how to devise efficient and correct programs. Once matters get to implementation in Maple, however, a typical scenario might be as follows:

1. You code the solution in the Maple programming language.
2. You enter the code into a text window or the **Maple Session** window.
3. Using the mouse, select the text and press enter.
4. With or without the program tracing tools `printlevel` and `trace` (described below), you test the code. You may have to go back and rethink the algorithm and the code if you detect an error during testing.
5. You achieve the desired result.

In this section, we discuss the tools available for Maple program development. We assume that by consulting the manuals for your computer system and by studying the previous section of this tutorial (or by referring to MRM), you have completed steps 1 and 2.

`printlevel` and **`trace`** are designed to provide step-by-step details on the execution of a Maple program as it is executing. They can be used to pinpoint where execution "diverges" from the programmer's notion of correctness.

`printlevel` is the simpler of the two tools. It is a global variable (default value 1) that controls the amount of information printed out during execution. This information includes assignments, expressions, and procedure entries and exits performed during program execution.

`printlevel` must be assigned an integer value. At its default value, it displays only the results of statements typed into Maple and statements nested one level inside `for-while` loops and `if` statements. If it is assigned zero, the results of statements nested one or more levels are not displayed, and only direct statements typed into the session are displayed. If it is assigned a negative value, no results (not even

those of simple commands) are displayed, except the output of `print` commands. The larger the value of `printlevel`, the more information is printed out on the results of function evaluations and other Maple statements nested within `for-while` statements, `if` statements, and procedures. The information includes procedure entries (showing the arguments passed), procedure exits (showing the value returned), and the results of executing statements nested up to the specified level. The nesting of statements is counted in the sense of nestings within `for-while` statements and `if` statements as well as nestings within procedures. Consult MRM or **`help('printlevel')`** for a fuller explanation.

Example 86

Programming aids: `printlevel`

```
# Example of printlevel set high enough to print out all intermediate
# results of a computation
>
   printlevel:=11:
# sum calls the built-in procedure `sum/indef`, which in turn calls
# other functions.
>
   sum(i^2,i=1..n);
--> enter sum, args = i**2, i = 1 .. n
                                    xx := i
                                     a := 1
                                   b := n + 1
                                    dab := n
                                    dxx := 2
--> enter sum/indef, args = i**2, i
>
   sum:indef: indefinite summation
<-- exit sum/indef = 1/3*i**3-1/2*i**2+1/6*i
                                  3       2
                            1/3 i  - 1/2 i  + 1/6 i
                                    3             2
                  1/3 (n + 1)  - 1/2 (n + 1)  + 1/6 n + 1/6
                                    3             2
                  1/3 (n + 1)  - 1/2 (n + 1)  + 1/6 n + 1/6
<-- exit sum = 1/3*(n+1)**3-1/2*(n+1)**2+1/6*n+1/6
                                    3             2
                  1/3 (n + 1)  - 1/2 (n + 1)  + 1/6 n + 1/6
# Reducing printlevel prints out information to a lesser depth of
# procedure calls.
>
   printlevel:=6:
>
   sum(i^3,i=1..n);
--> enter sum, args = i**3, i = 1 .. n
                                    xx := i
                                     a := 1
                                   b := n + 1
                                    dab := n
```

```
                                dxx := 3

sum:indef: indefinite summation
<— exit sum = 1/4*(n+1)**4-1/2*(n+1)**3+1/4*(n+1)**2
                                4          3          2
                      1/4 (n + 1)  - 1/2 (n + 1)  + 1/4 (n + 1)

# Reducing printlevel back to its default value of 1 returns things to
# normal:  just the results of the computation are printed.
>
   printlevel:=1;
                                printlevel := 1

>
   sum(i^4,i=1..n);
                        5          4          3
          1/5 (n + 1)  - 1/2 (n + 1)  + 1/3 (n + 1)  - 1/30 n - 1/30
```

trace(*procedure name*) provides a way of looking only at the results of particular procedures. A traced procedure will behave as if `printlevel` were set sufficiently high as to print all the statements and entry and exit points of that procedure.

To turn off tracing of a procedure, use **untrace**(*procedure name*).

Example 87

Programming aids: `trace` and `untrace`

```
# This proc provides a naive implementation of the Fibonacci numbers:
#    fib(0) = 0, fib(1) = 1,  fib(n) = fib(n-1) + fib(n-2)
>
   fib := proc(n)
>
    if not type(n,integer) then RETURN(`fib(n)`) fi;
>
    if n < 0               then RETURN((-1)^(1-n)*fib(-n)) fi;
>
    if n = 0 or n = 1      then RETURN(n) fi;
>
    fib(n-1) + fib(n-2)
>
   end:
>
   fib(5);
                                    5

>
   fib(12);
                                   144

# Hmmm... it seems a bit slow.  Let's trace it.
>
   trace(fib);
                                   fib

>
   fib(5);
--> enter fib, args = 5
--> enter fib, args = 4
--> enter fib, args = 3
--> enter fib, args = 2
--> enter fib, args = 1
<-- exit fib = 1
--> enter fib, args = 0
<-- exit fib = 0
                                    1
```

```
<-- exit fib = 1
--> enter fib, args = 1
<-- exit fib = 1
                                        2

<-- exit fib = 2
--> enter fib, args = 2
--> enter fib, args = 1
<-- exit fib = 1
--> enter fib, args = 0
<-- exit fib = 0
                                        1

<-- exit fib = 1
                                        3

<-- exit fib = 3
--> enter fib, args = 3
--> enter fib, args = 2
--> enter fib, args = 1
<-- exit fib = 1
--> enter fib, args = 0
<-- exit fib = 0
                                        1

<-- exit fib = 1
--> enter fib, args = 1
<-- exit fib = 1
                                        2

<-- exit fib = 2
                                        5

<-- exit fib = 5
                                        5
# No wonder! It is recomputing values. Let's add option remember.
>
  fib := proc(n)
>
   option remember;
>
   if not type(n,integer) then RETURN( 'fib(n)') fi;
>
   if n < 0               then RETURN((-1)^(1-n)*fib(-n)) fi;
>
   if n = 0 or n = 1      then RETURN(n) fi;
>
   fib(n-1) + fib(n-2)
>
  end:
# Let's trace the new fib.
>
  trace(fib);
                                      fib

>
  fib(5);
--> enter fib, args = 5
--> enter fib, args = 4
--> enter fib, args = 3
--> enter fib, args = 2
--> enter fib, args = 1
<-- exit fib = 1
--> enter fib, args = 0
<-- exit fib = 0
                                        1
```

```
<— exit fib = 1
                                    2
<— exit fib = 2
                                    3
<— exit fib = 3
                                    5
<— exit fib = 5
                                    5
# OK. Much better. To turn off tracing, use untrace.
>
   untrace(fib);
                                   fib
>
   fib(20);
                                   6765
# P.S. A better version of the Fibonacci numbers is already in the library.
>
   fib := readlib(fibonacci):
>
   fib(1000);
434665576869374564356885276750406258025646605173717804024817290895365554179490\
518904038798400792551692959225930803226347752096896232398733224711616429964409\
065331879382989696499285160037044761377951668492288 75
```

5.16 Conclusion

We hope *First Leaves* has given you an understanding of how to use Maple's built-in functions, data structures, and programming language. We would like to hear your comments on Maple and on this tutorial. Please address correspondence to:

Brooks/Cole Publishing Company
511 Forest Lodge Road
Pacific Grove, California 93950

For those with access to electronic mail, our electronic addresses are

Internet maple@watmath.Uwaterloo.Ca
Applelink D2248
Compuserve 70014;2461

Happy computing!

APPENDIX

Symbols with Predefined Meanings in Maple

TABLE 13
Symbols with predefined meanings in Maple

abs[2]	break[3]	D[2]	evala[2]	GAMMA[2]
addressof[2]	bspline[2]	define[2]	evalapply[2]	gamma[3]
algebraic[3]	builtin[3]	degree[2]	evalb[2]	gc[2]
anames[2]	by[1]	denom[2]	evalc[2]	gcd[2]
and[1,3]	C[2]	diff[2]	evalf[2]	gcdex[2]
arccos[2]	cat[2]	difforms[2]	evalm[2]	genpoly[2]
arccosh[2]	Catalan[3]	Digits[4]	evaln[2]	grobner[2]
arccot[2]	chebyshev[2]	dilog[2]	exp[2]	group[2]
arccoth[2]	chrem[2]	dinterp[2]	expand[2]	has[2]
arccsc[2]	Ci[2]	disassemble[2]	factor[2]	hastype[2]
arccsch[2]	coeff[2]	discrim[2]	factorial[3]	help[2]
arcsec[2]	coeffs[2]	divide[2]	FAIL[3]	hypergeom[2]
arcsech[2]	coeftayl[2]	do[1]	false[3]	I[3]
arcsin[2]	collect[2]	done[1]	fi[1]	icontent[2]
arcsinh[2]	commutat[2]	dsolve[2]	fibonacci[2]	if[1]
arctan[2]	compoly[2]	E[3]	fixdiv[2]	ifactor[2]
arctanh[2]	constant[3]	edit[2]	Float[2]	igcd[2]
args[3]	constants[4]	Ei[2]	float[3]	igcdex[2]
array[2]	content[2]	elif[1]	for[1]	ilcm[2]
assemble[2]	convergs[2]	else[1]	forget[2]	in[1]
assign[2]	convert[2]	end[1]	fortran[2]	indets[2]
assigned[2]	copy[2]	eqn[2]	frac[2]	indexed[3]
asympt[2]	cos[2]	equation[3]	fraction[3]	indices[2]
atatsign[2]	cosh[2]	erf[2]	freeze[2]	infinity[3]
atsign[2]	cot[2]	ERROR[2]	from[1]	int[2]
bernoulli[2]	coth[2]	euler[2]	frontend[2]	integer[3]
binomial[2]	csc[2]	eulermac[2]	fsolve[2]	interp[2]
boolean[3]	csch[2]	eval[2]	function[3]	intersect[1]

continued

TABLE 13 (continued)
Symbols with predefined meanings in Maple

iquo[2]	mods[2]	preplot[4]	RootOf[2]	substring[2]
irem[2]	msolve[2]	prettyprint[4]	round[2]	sum[2]
iroot[2]	mtaylor[2]	prevprime[2]	rsolve[2]	symmpoly[2]
iscont[2]	name[3]	primpart[2]	S[2]	sysindets[2]
isolate[2]	nargs[3]	print[2]	save[1]	system[2]
isolve[2]	next[3]	printlevel[4]	screenheight[4]	table[2]
isprime[2]	nextprime[2]	priqueue[2]	screenwidth[4]	tan[2]
isqrt[2]	nops[2]	proc[1]	search[2]	tanh[2]
issqr[2]	normal[2]	procedure[3]	sec[2]	taylor[2]
ithprime[2]	not[2,3]	procname[3]	sech[2]	tcoeff[2]
laplace[2]	np[2]	product[2]	select[2]	testeq[2]
lasterror[3]	NULL[4]	profile[2]	series[3]	then[1]
latex[2]	numer[2]	prompt[4]	set[3]	time[2]
lcm[2]	numeric[3]	proot[2]	showtime[2]	to[1]
lcoeff[2]	numtheory[2]	Psi[2]	Si[2]	totorder[2]
ldegree[2]	O[2]	psqrt[2]	sign[2]	trace[2]
length[2]	od[1]	quit[1]	signum[2]	translate[2]
lexorder[2]	op[2]	quo[2]	simplex[2]	traperror[2]
libname[4]	operator[3]	radical[3]	simplify[2]	trigsubs[2]
limit[2]	optimize[2]	radsimp[2]	sin[2]	true[3]
linalg[2]	option[1]	rand[2]	singular[2]	trunc[2]
list[3]	options[1]	randpoly[2]	sinh[2]	type[2]
ln[2]	or[1,3]	range[3]	sinterp[2]	unames[2]
local[1]	order[2]	rational[3]	solve[2]	unapply[2]
log[2,4]	Order[4]	ratpoly[3]	sort[2]	uneval[3]
lprint[2]	orthopoly[2]	read[1]	sprem[2]	union[1,3]
map[2]	permute[2]	readlib[2]	sqrt[2]	unknown[3]
match[2]	Pi[3]	readstat[2]	stats[2]	whattype[2]
max[2]	plot[2]	realroot[2]	status[4]	while[1]
maxnorm[2]	plotdevice[4]	relation[3]	stirl[2]	with[2]
member[2]	plotout[4]	rem[2]	stir2[2]	words[2]
min[2]	pointto[2]	remember[2]	stop[1]	write[2]
minimize[2]	polynom[3]	replot[2]	string[3]	writeto[2]
minus[1]	postplot[4]	residue[2]	student[2]	Zeta[2]
mod[1,4]	powseries[2]	resultant[2]	subs[2]	zip[2]
modp[2]	prem[2]	RETURN[2]	subsop[2]	

The symbols come from the following sources:

1. Parts of the Maple command/programming language such as `for`, `done`, or `if`. These symbols are impossible to redefine. They can only be used in Maple as the part of speech specified by the grammar of the Maple language—e.g., `then` must only

appear in `if` statements, `while` in repetitions, etc. This is the reason why `help(`if`)` works, but `help(if)` doesn't.

2. Maple system-supplied functions. These are functions such as `solve` or `int`. They may be redefined by the user's programming, but this will mean the loss of the functionality of the system-supplied function.

3. Maple predefined constants and names, such as `I`, `true`, or `list`. Various library functions assume that such symbols have no assigned value, with erroneous results ensuing otherwise.

4. Maple system programming variables, such as `Digits` and `printlevel`. These are assigned values when a Maple session is started, to control various computational effects (e.g., the number of digits carried in floating-point numbers, the amount of debugging information displayed as a computation is running, etc.). Changing the values of these variables should be done with caution. They should be used only for their intended system-controlling purpose.

Many of these symbols become defined or used only when a library file is loaded into a Maple session. Consult MRM or `help` for further details.

Index

Special characters

#, 3, 14
^, 12
{}, 81
$, 80
!, 12
", 12
', 70
*, 12
**, 12
+, 12
_, in variable names, 23
-, 12
., 56
/, 12
:, 12, 56
:, in folder name, 116
:=, 22
;, 12
  as statement separator, 105
`, 22, 50
`.`, 80

A

abs, 12, 26
absolute value, 12
addition, 12
algebraic numbers, 33
Algol 68, 100
apostrophe, 23, 69
arccos, 26
arccosh, 26
arccot, 26
arccoth, 26
arccsc, 26
arccsch, 26
arcsec, 26
arcsech, 26
arcsin, 26
arcsinh, 26
arctan, 26
arctanh, 26
args, 107
argument of a function, 28, 103
array, 80
arrays, 69
assign, 32
assigned, 69
assignment, 22, 71
asympt, 38
asymptotic series, 38

B

backquote, 22, 59
  in long names, 50
Bernoulli numbers, 25
Bessel functions, 26
bidirectional limit, 41
binomial coefficient, 26
brackets
  (| and |), 82
  [and], 82
break, 102
built-in commands, 3
built-in mathematics, 25
by, 98

C

C, 26
calculator numbers, 42
calculus package, 95
call, function, 103
case, lower and upper, 22

cases, 100
character string, 22
Ci, 26
Clear, 15
Clipboard, 16, 46
coeff, 78
colon, 12, 56
commands, 3, 27
comment lines, 14
comments, program, 3
common problem, 12–13, 50–51, 54, 69, 75, 98
complex infinity, 41
complex limit, 41
concatenation, 56, 80
content, integer, 66
convert, 117
copy, 87
Copy, 16, 46
cos, 26
cosh, 26
cosine integral, 26
cot, 26
coth, 26
csc, 26
csch, 26
cursor, 20
customizing Maple, 9
Cut, 16

D

data points, 46
data set, 53
decimal fraction numbers, 42
default folder, 115
degrees, 26
denom, 44
dialog, Out of stack space, 52
diff, 28, 35
differential equations, solution of, 35
differential forms, 95
differentiation, 28
Digits (default 10), 42
dilogarithm, 26
ditto, 12
divide, 72, 100, 104
division, 12
do, 96, 98
done, 14
double-quote, 12
dsolve, 35

E

E, 25
editing, 15
Edit menu, 6, 15
Ei, 26
elif, 100
else, 100
ending a session, 14
enter, 15, 19
eqn, 48
equation, 31
erf, 26
error
  run-time, 51
  syntax, 49
ERROR, 111
error function, 26
error message, from Maple, 51
escape sequences, for plotting on terminals, 46
Euler numbers, 25
evalb, 100
evalf, 43, 90
evalm, 93
evaln, 69
evaluate expression, 19
evaluation, 12
  full, 68
  last name, 69, 86, 90, 103
exception, to full evaluation, 69, 103, 106
exit
  from loop, 102
  from Maple session, 14
exp, 25
expand, 65
expanding products, 65
expandoff, 66
expandon, 66
explicit folder name, 116
explicit path name, 116
exponential integral, 26
exponentiation, 12, 14

expression sequence, 79–80, 82
exprseq, 80

F

factor, 67
factorial, 12, 26
factoring polynomials, 67
false, 25
FBAP languages, 61
fi, 100
Fibonacci numbers, 3, 53, 56, 110, 125
file, 53
  library object, 8
  MapleInit, 115
  Maple internal-format, 8, 53
  names, 116
  plot, 8, 45
  text, 8
File menu, 6
file name, conversion, 117
floating-point numbers, 25, 42
folder
  default, 116
  name, explicit, 116
  names, 116
Font menu, 19
for, 98–99, 102
for–from–by–to, 98
for–in, 99
fortran, 47
Fortran, idiosyncrasies, 47
Fresnel integrals, 26
from, 98
fsolve, 34, 36
full evaluation, 68
function, 28

G

GAMMA, 26
garbage collection, 20, 122
gc, 122
gcd, 26
graphing, of functions, 45
greatest common divisors, 26

H

H, 26
Halt menu, 20
harmonic numbers, 26
help, 57
help, 46, 56, 93, 95
help(inifcns), 26
help(ininame), 25
Help folder, 7
hypergeometric functions, 26

I

I, 25
icon, 6, 8–9, 17
if, 100
igcd, 97
in, 99
index
  for arrays and tables, 83
  for a solve solution, 31
  of a table, 84
infinity, 40
initialization file, 115
Initial library name..., 7
Initial stack size..., 53
installing Maple on the Macintosh, 5
int, 29
integer, *type*, 77
integers, 25
integration
  definite, 29
  indefinite, 29
interrupt, 20
intersect, 81
irem, 96
iroot, 57
isolve, 36

K

kernel, 3

L

labelling expressions, 22
latex, 48

launch Maple, 8
Laurent series, 38
lcm, 26
lcoeff, 78, 100
least common multiples, 26
left quote, 22, 50, 59
_liblist, 118
libname, 7, 56, 116
library, 3, 55
library object file, 8
limit, 40
limiting values of functions, 40
linalg, 92
linear algebra, 92
list, 82
ln, 26
log, 26
lprint, 24, 47

M

Maple II, 6
MapleInit file, 115
Maple internal-format file, 8, 53
Maple library, 5–7, 116
Maple plot file, 8, 45
Maple Plus, 6
Maple Session window, 16–17, 55
Maple text file, 8
mathematical
  symbol, 71
  variable, 22
matrices, 92
matrix operations, 93
max, 26
memory, 20, 120
  recycled, 122
  stack, 53
  usage, 120
min, 26
minus, 81
mod, 90
monomial, 63
msolve, 36
multiplication, 12
  matrix, 93

N

name, 80
names, 22
nargs, 107
negation, 14
New Folder menu, 6
next, 102
nops, 75, 101
normal, 66, 90
normalized expression, 66
NULL, 79
numbers, 25, 42
  algebraic, 34
number theory, 95
numer, 44

O

od, 96, 98
op, 36, 75, 79, 81
Open, 16, 47, 54
optimize, 47
option remember, 110
options, 110
Order (default 6), 38
ordering
  in sets, 81
  of terms within expressions, 63
orthogonal polynomials, 95
Out of stack space, 52

P

package, 95
  of functions, 92
parameter plots, 46
parentheses, 14
Pi, 25
PICT, 46
plot, 45
plotting
  of functions, 45
  several functions, 45
polar coordinates, 46
polynom, *type*, 77
postplot, 46
power series, 38
  manipulation package, 95

precedence of arithmetic operations, 13
preplot, 46
prettyprinting, 12, 24, 47
print, 24
printing, inside for-while loops, 123
printlevel, 97, 123
proc, 80
procedures, 69
programming variables, 22
prompt, 115
prompt (>), 19
Psi, Psi1, Psi2, etc., 26

Q

quit, 20
quit, 14
quotation marks, 12
  double, 12
  right, 70
  single, 69
quoting expressions, when to, 69

R

radians, 26
range, 41
rational numbers, 25
read, 54, 56
reading text files, 16
readlib, 47, 55
recursive definition, 52
reducing window width, 17
replacement, 36
resetting a variable, 71
restoring a session, 54
result of a function, 28
resume from a computation, 20
return, 11, 15–16, 19
RETURN, 108
Riemann's zeta function, 26
right quote, 23
RootOf, 34
roots of polynomial equations, 34
rsolve, 36
run-time error, 51
run-time stack, 52

S

S, 26
save
  Maple Session window, 55
  plot window, 46
  session, 53
  text window, 55
save, 47, 49, 53
Save, 55
Save As..., 46, 55
scroll bar, 17
sec, 26
sech, 26
Select All menu, 6
selection operation, 84
  for arrays and tables, 83
  for a set, list, or sequence, 82
  for a solve solution, 31
semicolon, 12
sequence, expression, 79, 82
series, Taylor, Laurent, and asymptotic, 38
session
  ending a, 14
  starting a, 11
set, 31, 81
  in Maple, 45
  ordering in, 81
set, 80
Settings menu, 7, 53
setup, for plotting mode, 46
showtime, 121
Si, 26
simplification, of expressions, 27, 34
sin, 26
sine integral, 26
sinh, 26
skipping statements in a loop, 102
solution, to algebraic equations, 31
solve, 31, 79
splines, 46
sqrt, 26
stack space, 53
  running out of, 52
starting a session, 11
start Maple, 8
startup screen, 9
statistics package, 95
Status window, 11, 120

Stirling numbers, 26
stop, 14
stopping a computation, 20
student package, 95
subs, 36
subscript
  list, 101
  of a table or array, 84
substitution, 36
subtraction, 12
sum, 30
summation, 30
symbol, 71
  mathematical, 62
syntax error, 49
System error message, 53

T

table, 69
  data structure, 83
  in Maple, 84
table, 80
tan, 26
tanh, 26
taylor, 38
Taylor series, 38
tcoeff, 78
Terminate current computation, 20
Terminate entire session, 20
then, 100
time, 121
time, usage, 120
to, 98
trace, 95, 125
trace, of a procedure, 123
translation, file name, 116
trigonometric functions, 26
true, 25
type, 75

U

uglyprinting, 47
unassigned variable, 22
unassignment, 71
uneval, 80
union, 81
unknowns, algebraic, 22
untrace, 125
updates, list of, to Maple, 60

V

value, of a name, 62
variable, 22
  global, 109
  mathematical, 22, 62
  programming, 22, 62
  quoted, 72
  unassigned, 22, 72
vector, 92
  operations, 93
versions, of Maple, 60
View menu, 6

W

Warning, Recursive definition, 52
warning message, from Maple, 50
watch hand, 20
whattype, 75
while, 96, 102
Window menu, 121
writing a file, 53

Z

zeroes of polynomials, 34
zip, 92
Zeta, 26

First Leaves for the Macintosh®

Step-by-step lessons introduce you to the most efficient symbolic computation system ever developed for the Macintosh. Using MAPLE and the *First Leaves* tutorial, novice and advanced users can now solve equations that were previously possible only on a mainframe.

Invaluable for those learning MAPLE, *First Leaves for the Macintosh* clearly explains how to use the software interactively as a calculator of formulas and numbers. For those who already have programming experience in another language, it also provides an introduction to writing and debugging programs using MAPLE's programming language. Clear, understandable examples make learning quick and memorable.

ISBN 0-534-10222-0

90000

9 780534 102227

First Leaves for the Macintosh was designed to complement the *MAPLE Reference Manual*, Fifth Edition.